Keep looking until you find it.

spring

★ 帮助读者**快速熟悉**Spring源码，以便对Spring源码进行**扩展**或**修改**，从而满足业务需求

★ 所有知识点均以**HelloWorld级别示例**为切入点，描述**简单**之后的复杂

★ 对于**复杂逻辑的讲解**采用**剥洋葱**似的方式，层层**分解复杂度**，便于读者理解和掌握

Spring 源码深度解析

● 郝佳 编著



人民邮电出版社
POSTS & TELECOM PRESS

Keep looking until you find it.

spring

★ 帮助读者**快速熟悉**Spring源码，以便对Spring源码进行**扩展**或**修改**，从而满足业务需求

★ 所有知识点均以**HelloWorld级别示例**为切入点，描述**简单**之后的复杂

★ 对于**复杂逻辑的讲解**采用**剥洋葱**似的方式，层层**分解复杂度**，便于读者理解和掌握

Spring
源码深度解析

● 郝佳 编著



人民邮电出版社
POSTS & TELECOM PRESS

Spring源码深度解析

郝佳　编著

人民邮电出版社

北京

前言

源代码的重要性

Java开发人员都知道，阅读源码是一个非常好的学习方式，在我们日常工作中或多或少都会接触一些开源代码，比如说最常用的Struts、Hibernate、Spring，这些源码的普及与应用程度远远超过我们的想象，正因为很多人使用，也在推动着源码不断地去完善。这些优秀的源码中有着多年积淀下来的精华，这些精华是非常值得我们学习的，不管我们当前是什么水平，通过反复阅读源码能力能有所提升，小到对源码所提供的功能上的使用更加熟练，大到使我们的程序设计更加完美优秀。但是，纵观我们身边的人，能够做到通读源码的真的是少之又少，究其原因不外乎以下几点。

阅读源码绝对算得上是一件费时费力的工作，需要读者耗费大量的时间去完成。而作为开发人员，毕竟精力有限，实在没办法拿出太多的时间放在源码的阅读上。

源码的复杂性。任何一款源码经历了多年的发展与提炼，其复杂程度可想而知。当我们阅读源码的时候，大家都知道需要通过工具来跟踪代码的运行，进而去分析程序。但是，当代码过于复杂，环环相扣绕来绕去的时候，跟进了几十个甚至几百个函数后，这时我们已经不知道自己所处的位置了，不得不再重来，但是一次又一次的，最终发现自己根本无法驾驭它，不得不放弃。

有些源码发展多年，会遇到各种各样的问题，并对问题进行了解决，而这些问题有的对于我们来说甚至可以用莫名其妙来修饰，有时候根本想不出会在什么情况下会发生。我们选择各种查阅资料，查询无果，失去耐心，最终放弃。

无论基于什么样的原因，放弃阅读源码始终不是一个明智的选择，

因为你失去了一个跟大师学习的机会。而且，当你读过几个源码之后你会发现，他们的思想以及实现方式是相通的。这就是开源的好处。随着各种开源软件的发展，各家都会融合别家优秀之处来不断完善自己，这样，到最后的结果就是所有的开源软件从设计上或者实现上都会变得越来越相似，也就是说当你读完某个优秀源码后再去读另一个源代码，速度会有很大提升。

以我为例，Spring是我阅读的第一个源码，几乎耗尽了我将近半年的时间，其中各种煎熬可想而知，但是当我读完Spring再去读MyBatis只用了两周时间。当然，暂且不论它们的复杂程度不同，至少我阅读的时候发现有很多相通的东西。当你第一次阅读的时候，你的重点一定是在源码的理解上，但是，当你读完第一个源码再去读下一个的时候，你自然而然地会带着批判或者说挑剔的眼光去阅读：为什么这个功能在我之前看的源码中是那样实现的，而在这里会是这样实现的？这其中的道理在哪里，哪种实现方式更优秀呢？而通过这样的对比及探索，你会发现，自己的进步快得难以想象。

我们已经有些纠结了，既然阅读源码有那么多的好处，但是很多同学却因为时间或者能力的问题而不得不放弃，岂不是太可惜？为了解决这个问题，我撰写了本书，总结了自己的研究心得和实际项目经验，希望能对正在Spring道路上摸索的同仁们提供一些帮助。

本书特点

本书完全从开发者的角度去剖析源码，每一章都会提供具有代表性的实例，并以此为基础进行功能实现的分析，而不是采取开篇就讲解什么容器怎么实现、AOP怎么实现之类的写法。在描述的过程中，本书尽可能地把问题分解，使用剥洋葱的方式一层一层地将逻辑描述清楚，帮助读者由浅入深地进行学习，并把这些难点和问题各个击破，而不是企图一下让读者理解一个复杂的逻辑。

在阅读源码的过程中，我们难免会遇到各种各样的生僻功能，这些

功能在特定的场合会非常有用，但是可能多数情况下并不是很常用，甚至都查阅不到相关的使用资料。本书中重点针对这种情况提供了相应的实用示例，让读者更加全面地了解Spring所提供的功能，对代码能知其然还知其所以然。

本书按照每章所提供的示例跟踪Spring源码的流程，尽可能保证代码的连续性，使读者的思维不被打乱，让读者看到Spring的执行流程，旨在尽量使读者在阅读完本书后即使在不阅读 Spring 源码的情况下也可以对 Spring 源码进行优化，甚至通过扩展源码来满足业务需求，这对开发人员来说是一个很高的要求。本书就希望能帮助读者全面提升实战能力。

本书结构

本书分为两部分：核心实现和企业应用。

第一部分 核心实现（第1～7章）：是Spring功能的基础，也是企业应用部分的基础，主要对容器以及AOP功能实现做了具体的分析，如果读者之前没有接触过 Spring 源代码，建议认真阅读这个部分，否则阅读企业应用部分时会比较吃力。

第二部分 企业应用（第8～13章）：在核心实现部分的基础上围绕企业应用常用的模块进行讨论，这些模块包括Spring整合JDBC、Spring整合MyBatis、事务、SpringMVC、远程服务、Spring 消息服务等，旨在帮助读者在日常开发中更加高效地使用Spring。

本书适用的**Spring**版本

截至完稿，Spring已经发布了 4.0.0.M1版本。本书虽然是基于Spring 3.2版本编写的，但所讨论的内容都属于Spring的基础和常用的功能，这些功能都经过长时间、大量用户的验证，已经非常成熟，改动的可能性相对较小。而且从目前Spring的功能规划来看，本书所涉及的内容并不在Spring未来改动的范围内，因此在未来的很长一段时间内本书都不会过时的。

感谢

创作的过程是痛苦的，持续时间也远远超乎了我的想象，而本以为自己对Spring已经非常的熟悉，但是在写作的过程中还是会遇到各种各样的问题，但是我很幸运我能坚持下来，在这里我首先应该感谢爸爸妈妈，虽然他们不知道儿子在忙忙碌碌地写些什么，但是他们对我始终如一的支持与鼓励使我更加坚定信心，在这里祝他们身体健康，同时还要感谢我最好的朋友孙亚超在我低落时给予我的关心与问候，当然要感谢的还有张雨绮同学，虽然并不是明星，但是却有着堪比明星般的美丽笑容，与她在一起的讨论总是让我受益匪浅，同时也感谢妹子王晶对稿件提供的建议与意见。最后感谢郭维云、郝云勃、郝俊、李兴全、梁晓颖、陈淼、孙伟超、王璐、刘瑞、单明、姚佳林、闫微微、李娇、时宇、李平、唐广亮、刘阳、黄思文、金施源等在整个编写过程中给予的支持与帮助。

联系作者

在编写本书过程中，以“够用就好”为原则，尽量覆盖到Spring开发的常用功能。所有观点都出自作者的个人见解，疏漏、错误之处在所难免，欢迎大家指正。读者如果有好的建议或者学习本书过程中遇到问题，请发送邮件到haojia_007@163.com，希望能够与大家一起交流和进步。

在看得见的地方学习知识，在看不到的地方学习智慧。祝愿大家在Spring的学习道路上顺风顺水。

作者
2013年7月

作者简介



郝佳

计算机专业硕士学位，曾发表过多篇论文先后被 EI、SCI 收录，2008 年辽宁省教育厅科技计划项目研究人之一；长期奋斗于 J2EE 领域，曾任职于某互联网公司软件架构师，擅长系统的性能优化，目前正投身于开发一款基于Java并发多线程管理的开源框架；热衷于研究各种优秀的开源代码并从中进行总结，从而实现个人技能的提高，尤其对Spring、Hibernate、MyBatis、JMS、Tomcat等源码有着深刻的理解和认识。

目　录

第一部分　核心实现



第1章　Spring整体架构和环境搭建

Spring是于 2003 年兴起的一个轻量级的Java开源框架，由Rod Johnson在其著作《Expert One-On-One J2EE Development and Design》中阐述的部分理念和原型衍生而来。Spring是为了解决企业应用开发的复杂性而创建的，它使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。

1.1 Spring的整体架构

Spring框架是一个分层架构，它包含一系列的功能要素，并被分为大约20个模块，如图1-1所示。

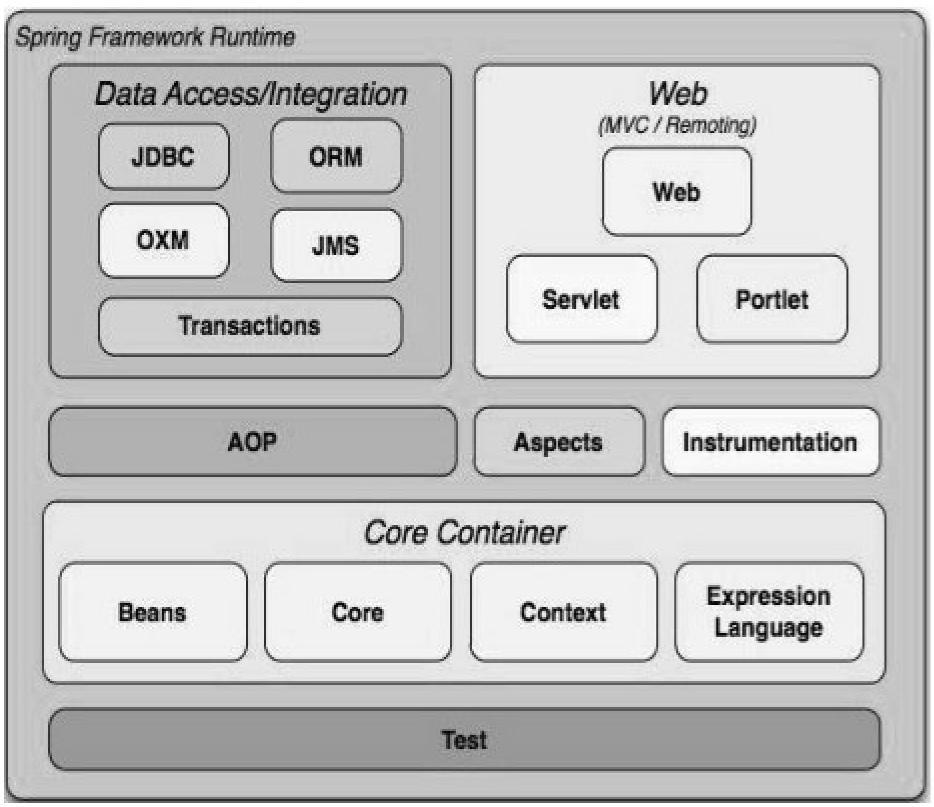


图1-1 Spring整体架构图

这些模块被总结为以下几部分。

（1）Core Container。

Core Container（核心容器）包含有Core、Beans、Context和Expression Language模块。

Core和Beans模块是框架的基础部分，提供IoC（转控制）和依赖注入特性。这里的基础概念是BeanFactory，它提供对Factory模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

Core模块主要包含Spring框架基本的核心工具类，Spring的其他组件要都要使用到这个包里的类，Core模块是其他组件的基本核心。当然你也可以在自己的应用系统中使用这些工具类。

Beans模块是所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行Inversion of Control / Dependency Injection（IoC/DI）操作相关的所有类。

Context模块构建于Core和Beans模块基础之上，提供了一种类似于JNDI注册器的框架式的对象访问方法。Context模块继承了 Beans的特性，为Spring核心提供了大量扩展，添加了对国际化（例如资源绑定）、事件传播、资源加载和对Context的透明创建的支持。Context模块同时也支持J2EE的一些特性，例如EJB、JMX和基础的远程处理。ApplicationContext接口是Context模块的关键。

Expression Language 模块提供了一个强大的表达式语言用于在运行时查询和操纵对象。它是 JSP 2.1规范中定义的 unifed expression language的一个扩展。该语言支持设置/获取属性的值，属性的分配，方法的调用，访问数组上下文（accessiong the context of arrays）、容器和索引器、逻辑和算术运算符、命名变量以及从Spring的 IoC容器中根据名称检索对象。它也支持list投影、选择和一般的list聚合。

（2）Data Access/Integration。

Data Access/Integration层包含有 JDBC、ORM、OXM、JMS和Transaction模块，其中：

JDBC模块提供了一个JDBC抽象层，它可以消除冗长的JDBC编码和解析数据库厂商特有的错误代码。这个模块包含了Spring对JDBC数据访问进行封装的所有类。

ORM模块为流行的对象-关系映射API，如JPA、JDO、Hibernate、iBatis等，提供了一个交互层。利用ORM封装包，可以混合使用所有Spring提供的特性进行O/R映射。如前边提到的简单声明性事物管理。

Spring框架插入了若干个ORM框架，从而提供了ORM的对象关系工具，其中包括JDO、Hibernate和iBatisSQL Map。所有这些都遵从Spring的通用事务和DAO异常层次结构。

OXM模块提供了一个对Object/XML映射实现的抽象层，Object/XML映射实现包括JAXB、Castor、XMLBeans、JiBX和XStream。

JMS（Java Messaging Service）模块主要包含了一些制造和消费消息的特性。

Transaction 模块支持编程和声明性的事物管理，这些事物类必须实现特定的接口，并且对所有的POJO都适用。

（3）Web。

Web上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文。所以， Spring框架支持与Jakarta Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。Web层包含了Web、Web-Servlet、Web-Struts和Web-Porlet模块，具体说明如下。

Web模块：提供了基础的面向Web的集成特性。例如，多文件上传、使用 servlet listeners初始化IoC容器以及一个面向Web的应用上下文。它还包含Spring远程支持中Web的相关部分。

Web-Servlet模块web.servlet.jar：该模块包含Spring的model-view-controller（MVC）实现。Spring的MVC框架使得模型范围内的代码和web forms之间能够清楚地分离开来，并与Spring框架的其他特性集成在一起。

Web-Struts模块：该模块提供了对Struts的支持，使得类在Spring应用中能够与一个典型的Struts Web层集成在一起。注意，该支持在Spring 3.0中是 deprecated的。

Web-Porlet模块：提供了用于Portlet环境和Web-Servlet模块的MVC的实现。

（4）AOP。

AOP模块提供了一个符合AOP联盟标准的面向切面编程的实现，它让你可以定义例如方法拦截器和切点，从而将逻辑代码分开，降低它们之间的耦合性。利用 source-level 的元数据功能，还可以将各种行为信息合并到你的代码中，这有点像.Net技术中的attribute概念。

通过配置管理特性，Spring AOP模块直接将面向切面的编程功能集成到了Spring框架中，所以可以很容易地使Spring框架管理的任何对象支持AOP。Spring AOP模块为基于Spring的应用程序中的对象提供了事务管理服务。通过使用Spring AOP，不用依赖EJB组件，就可以将声明性事务管理集成到应用程序中。

Aspects模块提供了对AspectJ的集成支持。

Instrumentation模块提供了 class instrumentation支持和 classloader实现，使得可以在特定的应用服务器上使用。

（5）Test。

Test模块支持使用JUnit和TestNG对Spring组件进行测试。

1.2 环境搭建

Spring已经将源码从 svn迁移到了GitHub。而且也改为基于Gradle的构建来构建项目，它取代了之前的Ant+Ivy系统，所以要构建Spring源码环境首先要安装GitHub以及Gradle。

1.2.1 安装GitHub

首先读者需要到GitHub官网去下载安装包，其中Windows系统对应的版本下载地址为：http://windows.github.com/，下载后双击进行安装。安装成功后，快捷菜单中会出现 GitHub 的菜单，如图1-2所示。



图1-2 HitHub安装成功后的启动菜单

1.2.2 安装Gradle

Gradle是一个基于Groovy的构建工具，它使用Groovy来编写构建脚本，支持依赖管理和多项目创建，类似 Maven，但比其更加简单轻便。Gradle 为 Ivy 提供了一个 layer，提供了build-by-convention 集成，而且它还让你获得许多类似 Maven 的功能。你可以从 http://www.gradle.org/downloads 页面下载 Gradle ，下载后将文件解压放到指定目录中（笔者放在了C:\Program Files目录下），然后开始进行环境变量的配置。

（1）根据对应目录创建GRADLE_HOME系统变量，如图1-3所

示。

（2）将系统变量加入到path中，如图1-4所示。



图1-3创建对应于Gralde的系统变量



图1-4将Gradle对应的系统变量加入path中

（3）测试。

当完成系统变量的配置后打开命令窗口输入命令“gradle –version”，如果安装成功会出现Gradle对应的版本信息，如图1-5所示。



图1-5 测试Gradle的环境变量配置

1.2.3 下载Spring

因为Spring源码是通过GitHub进行管理的，所以我们首先打开

GitHub，单击快捷菜单中的“Git Shell”选项，如图 1-6所示。

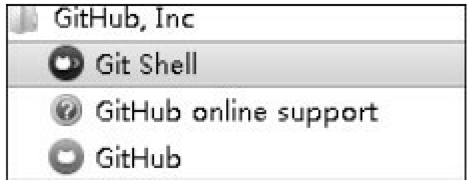


图1-6 启动GitHub启动菜单

打开GitHub后，你可以通过cd命令将当前操作目录转换到我们想要存储源码的目录，例如，想要将下载的源码存储到e:\test下，则可以执行“cd e:\test”。

输入以下命令：

git clone git://github.com/SpringSource/Spring-framework.git

其中“git://github.com/SpringSource/Spring-framework.git”为Spring的源码地址。执行命令后便进入源码下载状态，如图1-7所示。



图1-7 使用GitHub开始下载源码

经过一段时间的等待后源码下载结束，窗口状态如图1-8所示。



图1-8 源码下载结束的窗口显示

而这时候我们去查看，对应的文件夹下已经存在了相应的源码信息，如图1-9所示。

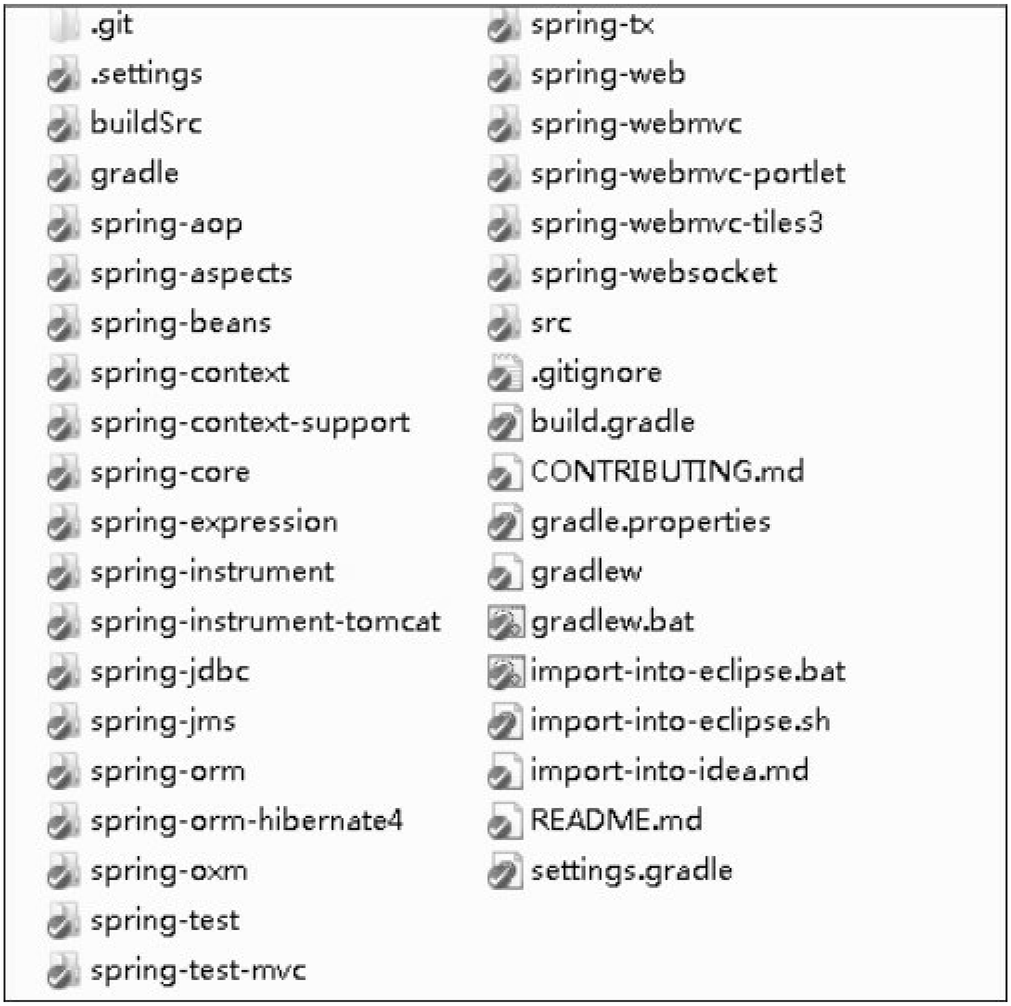


图1-9 下载的Spring源码

但是当前的源码并不可以直接导入Eclipse中，我们还需要将源码转换为Eclipse可以读取的形式。网上有各种各样的方法，其中出现最多的是告诉大家将所有工程一次性的编译、导入，但是笔者并不推荐这样的方式，因为这样会耗费大量的时间，而且当存在编译错误的时候你不得不重新编译。笔者建议只对我们感兴趣的工程进行 Eclipse 工程转换，比如我们想要查看Spring事务部分的源码，打开命令窗口，将当前目录切换至源码所在目录，例如，这里是Spring-tx文件夹下，执行命令“gradle cleanidea eclipse”，当窗口出现如下状态说明已经开始执行转换过程，如图1-10所示。

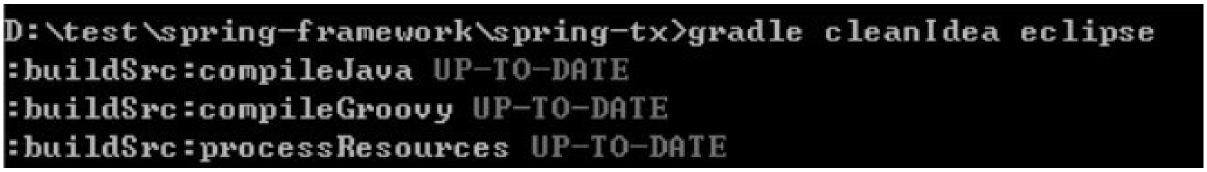


图1-10 Spring源码转换至eclipse工程

经过一段时间后转换成功，如图1-11所示。

这时，我们再查看对应的文件夹会发现，已经出现了作为 Eclipse 工程所必须的.project与.classpath文件了，如图1-12所示。

打开Eclipse，将工程导入，导入后如图1-13所示。

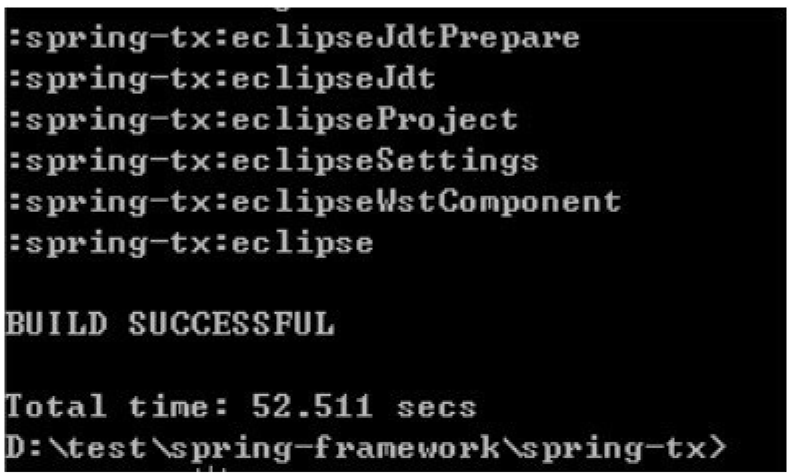


图1-11 Spring源码成功转换至eclipse工程

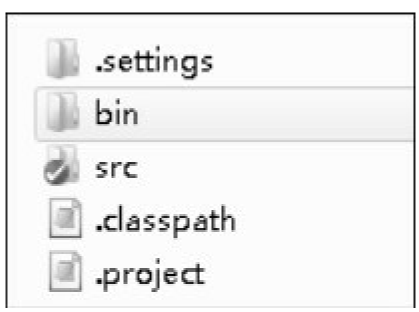


图1-12转换至Eclipse工程后的Spring源码结构

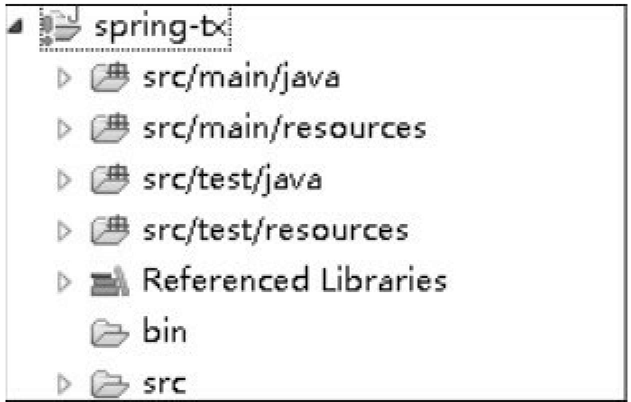


图1-13导入Eclipse后的源码工程

你会发现工程名称前面有一个感叹号，这说明存在错误。查看依赖包及工程，会看到当前工程所依赖的包已经完全导入，没有问题，工程所依赖的JAR如图1-14所示。

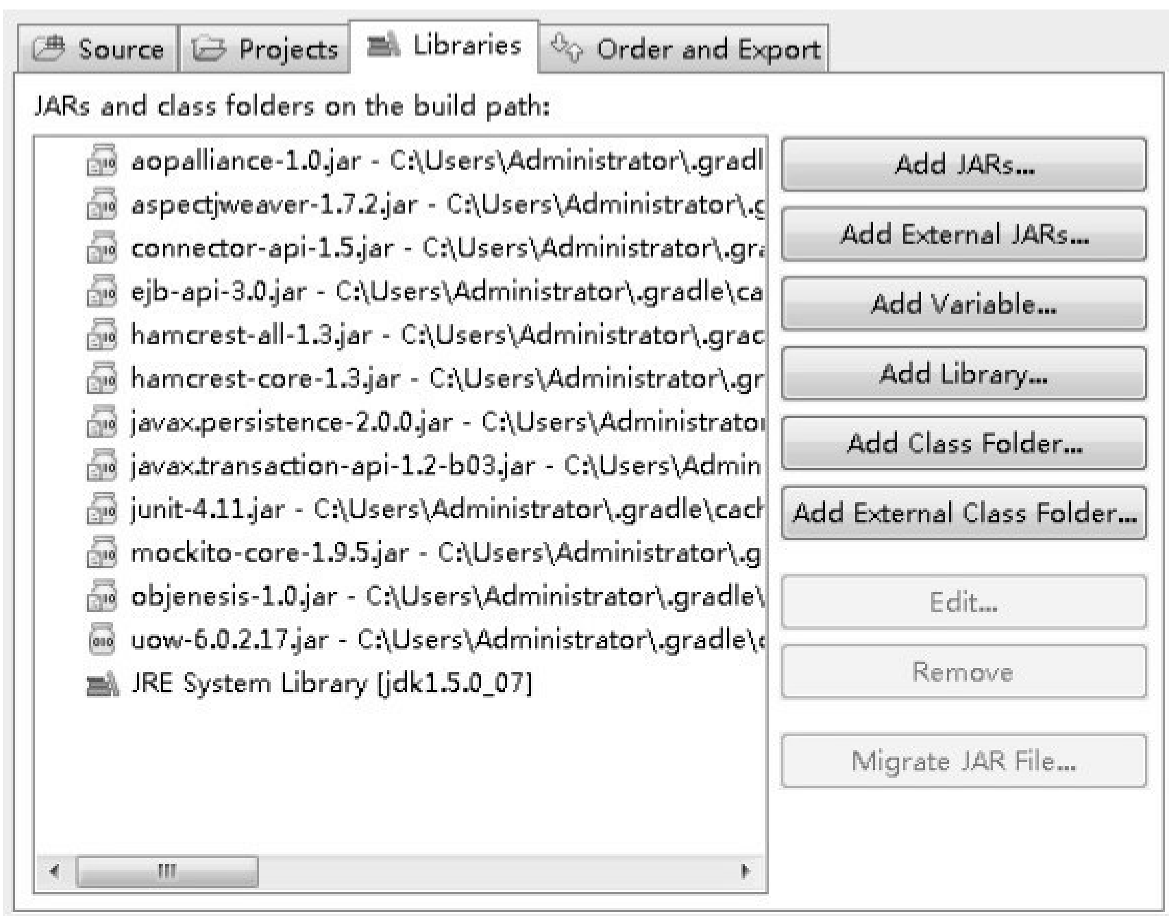


图1-14 工程依赖的JAR

但是，查看依赖的Projects时发现，当前工程还要依赖于其他Spring中的6个工程，这时，读者可以选择以同样的方式继续导入源码工程，或者，直接找到对应的JAR加入编译路径，工程所依赖的Projects如图1-15所示。

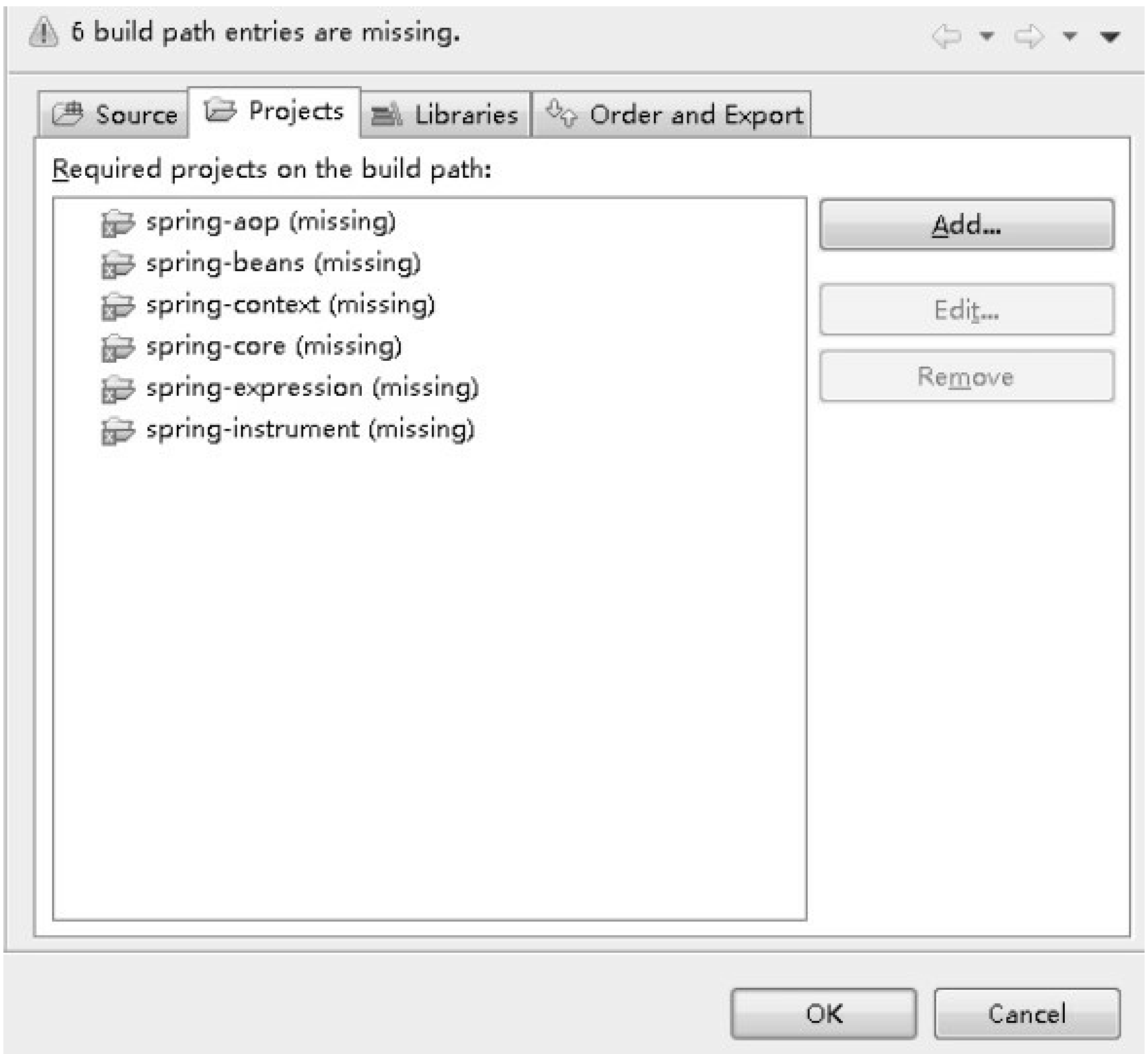


图1-15 工程所依赖的Projects

第2章　容器的基本实现

源码分析是一件非常煎熬非常有挑战性的任务，你准备好开始战斗了吗？

在正式开始分析 Spring 源码之前，我们有必要先来回顾一下 Spring 中最简单的用法，尽管我相信您已经对这个例子非常熟悉了。

2.1 容器基本用法

bean是Spring中最核心的东西，因为Spring就像是个大水桶，而bean就像是容器中的水，水桶脱离了水便也没什么用处了，那么我们先看看bean的定义。

```
public class MyTestBean {
    private String testStr = "testStr";
    public String getTestStr() {
        return testStr;
    }
    public void setTestStr(String testStr) {
        this.testStr = testStr;
    }
}
```

很普通，bean没有任何特别之处，的确，Spring的目的就是让我们的bean能成为一个纯粹的POJO，这也是Spring所追求的。接下来看看配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www. Springframework.
      org/schema/beans/Spring-beans.xsd">
       <bean id="myTestBean" class="bean.MyTestBean"/>
    </beans>
```

在上面的配置中我们看到了bean的声明方式，尽管Spring中bean的元素定义着N种属性来支撑我们业务的各种应用，但是我们只要声明成这样，基本上就已经可以满足我们的大多数应用了。好了，你可能觉得还有什么，但是，真没了，Spring的入门示例到这里已经结束，我们可以写测试代码测试了。

```
@SuppressWarnings("deprecation")
public class BeanFactoryTest {
  @Test
   public void testSimpleLoad(){
      BeanFactory bf = new XmlBeanFactory(new ClassPathResource (
"beanFactoryTest.xml"));
      MyTestBean bean=(MyTestBean) bf.getBean("myTestBean");
     assertEquals("testStr",bean.getTestStr());
  }
}
```

相信聪明的读者会很快看到我们期望的结果：在Eclipse中显示了Green Bar。

直接使用BeanFactory作为容器对于Spring的使用来说并不多见，甚至是甚少使用，因为在企业级的应用中大多数都会使用的是ApplicationContext（后续章节我们会介绍它们之间的区别），这里只是用于测试，让读者更快更好地分析Spring的内部原理。

OK，我们又复习了一遍Spring，你是不是会很不屑呢？这样的小例

子没任何挑战性。嗯，确实，这样的使用是过于简单了，但是本书的目的并不是介绍如何使用Spring，而是帮助您更好地了解 Spring 的内部原理。读者可以自己先想想，上面的一句简单代码都执行了什么样的逻辑呢？这样一句简单代码其实在Spring中执行了太多太多的逻辑，即使笔者用半本书的文字也只能介绍它的大致原理。那么就让我们快速的进入分析状态吧。

2.2 功能分析

现在我们可以来好好分析一下上面测试代码的功能，来探索上面的测试代码中Spring究竟帮助我们完成了什么工作？不管之前你是否使用过Spring，当然，你应该使用过的，毕竟本书面用的是对Spring有一定使用经验的读者，你都应该能猜出来，这段测试代码完成的功能无非就是以下几点。

（1）读取配置文件beanFactoryTest.xml。

（2）根据 beanFactoryTest.xml 中的配置找到对应的类的配置，并实例化。

（3）调用实例化后的实例。

为了更清楚地描述，笔者临时画了设计类图，如图2-1所示，如果想完成我们预想的功能，至少需要3个类。

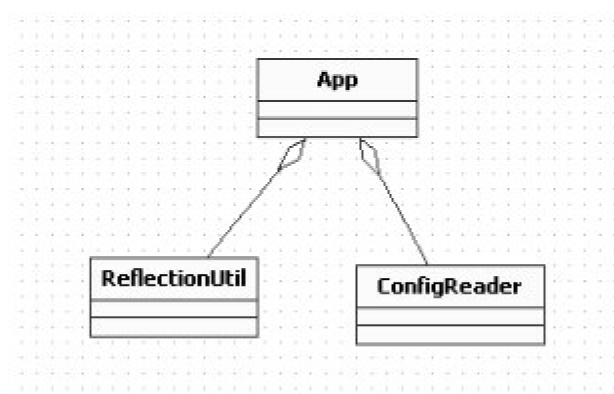


图2-1 最简单的Spring功能架构

ConfigReader：用于读取及验证配置文件。我们要用配置文件里面的东西，当然首先要做的就是读取，然后放置在内存中。

ReflectionUtil：用于根据配置文件中的配置进行反射实例化。比如在上例中beanFactoryTest.xml出现的<bean id="myTestBean" class="bean.MyTestBean"/>，我们就可以根据bean.MyTestBean进行实例化。

App：用于完成整个逻辑的串联。

按照最原始的思维方式，整个过程无非如此，但是作为一个风靡世界的优秀源码真的就这么简单吗?

2.3 工程搭建

不如我们首先大致看看 Spring 的源码。在 Spring 源码中，用于实现上面功能的是org.Springframework.beans.jar，我们看源码的时候要打开这个工程，如果我们只使用上面的功能，那就没有必要引入Spring的其他更多的包，当然Core是必须的，还有些依赖的包如图2-2所示。

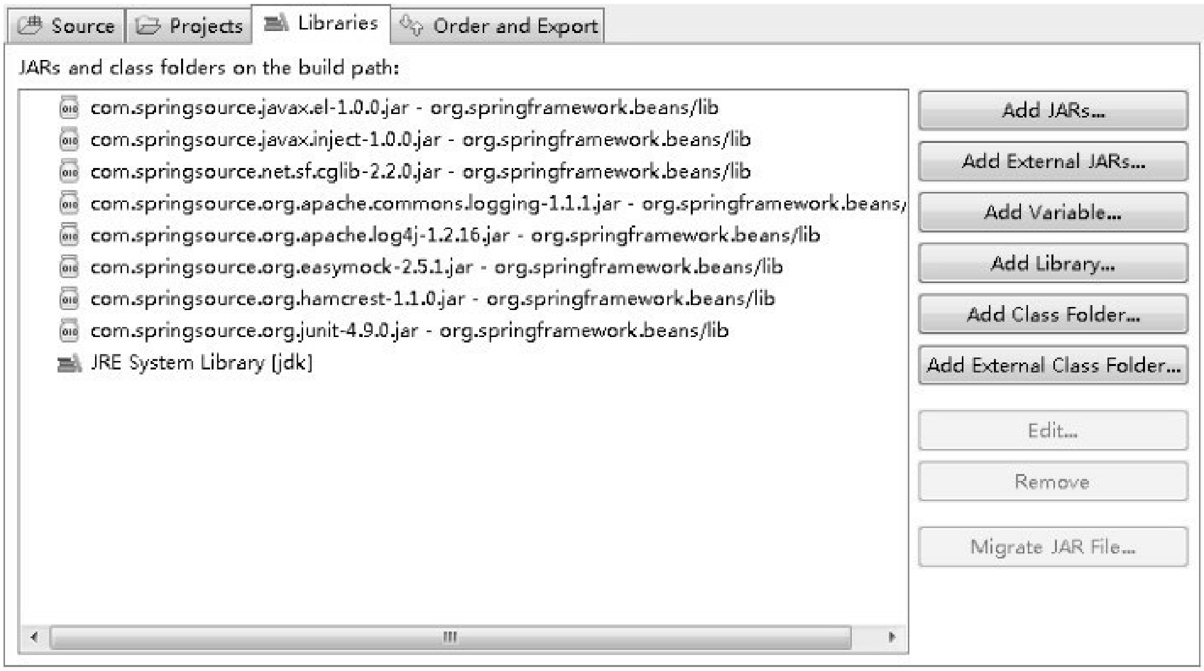


图2-2 Spring测试类依赖的JAR

引入依赖的 JAR 消除掉所有编译错误后，终于可以看源码了。或许你已经知道了答案， Spring 居然用了 N 多代码实现了这个看似很简单的功能，那么这些代码都是做什么用的呢？Spring在架构或者编码的时候又是如何考虑的呢？带着疑问，让我们踏上了研读Spring源码的征程。

2.4 Spring的结构组成

我们首先尝试梳理一下Spring的框架结构，从全局的角度了解一下Spring的结构组成。

2.4.1 beans包的层级结构

笔者认为阅读源码的最好方法是通过示例跟着操作一遍，虽然有时候或者说大多数时候会被复杂的代码绕来绕去，绕到最后已经不知道自己身在何处了，但是，如果配以UML还是可以搞定的。笔者就是按照

自己的思路进行分析，并配合必要的UML，希望读者同样可以跟得上思路。

我们先看看整个beans工程的源码结构，如图2-3所示。



图2-3 beans工程的源码结构

beans包中的各个源码包的功能如下。

src/main/java用于展现Spring的主要逻辑。

src/main/resources用于存放系统的配置文件。

src/test/java用于对主要逻辑进行单元测试。

src/test/resources用于存放测试用的配置文件。

2.4.2 核心类介绍

通过beans工程的结构介绍，我们现在对beans的工程结构有了初步的认识，但是在正式开始源码分析之前，有必要了解一下Spring中最核心的两个类。

1．DefaultListableBeanFactory

XmlBeanFactory继承自DefaultListableBeanFactory，而DefaultListableBeanFactory是整个bean加载的核心部分，是 Spring 注册及加载 bean 的默认实现，而对于 XmlBeanFactory 与 DefaultListableBeanFactory 不同的地方其实是在 XmlBeanFactory 中使用了自定义的 XML 读取器XmlBeanDefinitionReader，实现了个性化的BeanDefinitionReader读取，DefaultListableBeanFactory继承了AbstractAutowireCapableBeanFactory 并实现了ConfigurableListableBeanFactory 以及BeanDefinitionRegistry接口。以下是ConfigurableListableBeanFactory的层次结构图（见图2-4）以及相关类图（见图2-5）。

从上面的类图以及层次结构图中，我们可以很清晰地从全局角度了解 DefaultListableBean Factory的脉络。如果读者没有了解过Spring源码可能对上面的类图不是很理解，不过没关系，通过后续的学习，你会逐渐了解每个类的作用。那么，让我们先简单地了解一下上面类图中的各个类的作用。

AliasRegistry：定义对alias的简单增删改等操作。

SimpleAliasRegistry：主要使用map作为alias的缓存，并对接口AliasRegistry进行实现。

SingletonBeanRegistry：定义对单例的注册及获取。

BeanFactory：定义获取bean及bean的各种属性。

DefaultSingletonBeanRegistry：对接口SingletonBeanRegistry各函数的实现。

HierarchicalBeanFactory：继承BeanFactory，也就是在BeanFactory定义的功能的基础上增加了对parentFactory的支持。

BeanDefinitionRegistry：定义对BeanDefinition的各种增删改操作。

FactoryBeanRegistrySupport：在DefaultSingletonBeanRegistry基础上

增加了对FactoryBean的特殊处理功能。



图2-4 ConfigurableListableBeanFactory的层次结构图



图2-5 容器加载相关类图

ConfigurableBeanFactory：提供配置Factory的各种方法。

ListableBeanFactory：根据各种条件获取bean的配置清单。

AbstractBeanFactory：综合 FactoryBeanRegistrySupport和ConfigurableBeanFactory的功能。

AutowireCapableBeanFactory：提供创建bean、自动注入、初始化以

及应用bean的后处理器。

AbstractAutowireCapableBeanFactory：综合AbstractBeanFactory并对接口Autowire Capable BeanFactory进行实现。

ConfigurableListableBeanFactory：BeanFactory配置清单，指定忽略类型及接口等。

DefaultListableBeanFactory：综合上面所有功能，主要是对Bean注册后的处理。

XmlBeanFactory对DefaultListableBeanFactory类进行了扩展，主要用于从XML文档中读取BeanDefinition，对于注册及获取Bean都是使用从父类DefaultListableBeanFactory继承的方法去实现，而唯独与父类不同的个性化实现就是增加了XmlBeanDefinitionReader类型的reader属性。在XmlBeanFactory中主要使用reader属性对资源文件进行读取和注册。

2．XmlBeanDefinitionReader

XML配置文件的读取是Spring中重要的功能，因为Spring的大部分功能都是以配置作为切入点的，那么我们可以从XmlBeanDefinitionReader中梳理一下资源文件读取、解析及注册的大致脉络，首先我们看看各个类的功能。

ResourceLoader：定义资源加载器，主要应用于根据给定的资源文件地址返回对应的Resource。

BeanDefinitionReader：主要定义资源文件读取并转换为BeanDefinition的各个功能。

EnvironmentCapable：定义获取Environment方法。

DocumentLoader：定义从资源文件加载到转换为Document的功能。

AbstractBeanDefinitionReader：对EnvironmentCapable、BeanDefinitionReader类定义的功能进行实现。

BeanDefinitionDocumentReader：定义读取Docuemnt并注册

BeanDefinition功能。

BeanDefinitionParserDelegate：定义解析Element的各种方法。

经过以上分析，我们可以梳理出整个XML配置文件读取的大致流程，如图2-6所示，在XmlBeanDifinitionReader中主要包含以下几步的处理。

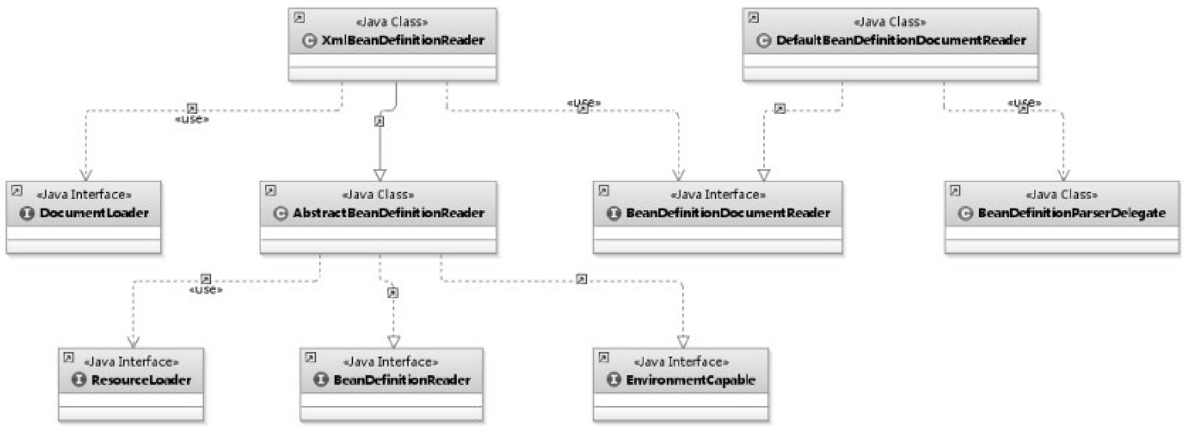


图2-6 配置文件读取相关类图

（1）通过继承自AbstractBeanDefinitionReader中的方法，来使用ResourLoader将资源文件路径转换为对应的Resource文件。

（2）通过DocumentLoader对Resource文件进行转换，将Resource文件转换为Document文件。

（3）通过实现接口BeanDefinitionDocumentReader的DefaultBeanDefinitionDocumentReader类对Document进行解析，并使用BeanDefinitionParserDelegate对Element进行解析。

2.5 容器的基础XmlBeanFactory

好了，到这里我们已经对Spring的容器功能有了一个大致的了解，尽管你可能还很迷糊，但是不要紧，接下来我们会详细探索每个步骤的

实现。再次重申一下代码，我们接下来要深入分析以下功能的代码实现：

```
BeanFactory bf = new XmlBeanFactory(new ClassPathResource("beanFactoryTest.xml"));
```

通过XmlBeanFactory初始化时序图（如图2-7所示）我们来看一看上面代码的执行逻辑。

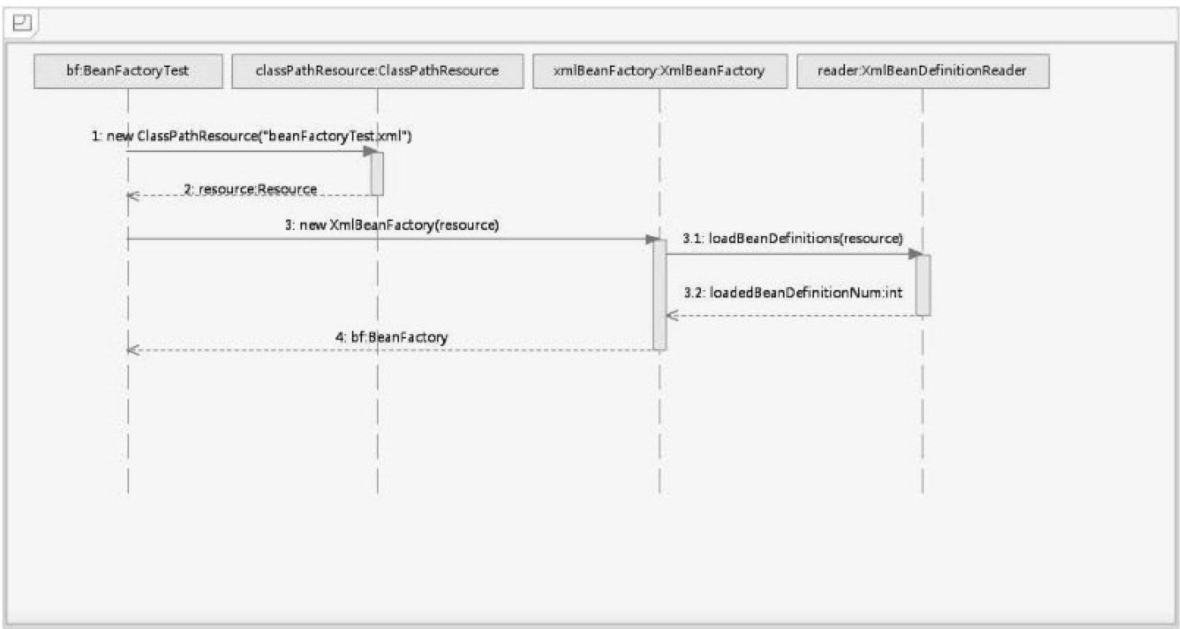


图2-7 XmlBeanFactory初始化时序图

时序图从BeanFactoryTest测试类开始，通过时序图我们可以一目了然地看到整个逻辑处理顺序。在测试的BeanFactoryTest中首先调用ClassPathResource的构造函数来构造Resource资源文件的实例对象，这样后续的资源处理就可以用Resource提供的各种服务来操作了，当我们有了Resource后就可以进行XmlBeanFactory的初始化了。那么Resource资源是如何封装的呢？

2.5.1 配置文件封装

Spring的配置文件读取是通过 ClassPathResource进行封装的，如new ClassPathResource ("beanFactoryTest.xml")，那么ClassPathResource完成了什么功能呢？

在Java中，将不同来源的资源抽象成URL，通过注册不同的handler（URLStreamHandler）来处理不同来源的资源的读取逻辑，一般handler的类型使用不同前缀（协议，Protocol）来识别，如“file:”、“http:”、“jar:”等，然而 URL 没有默认定义相对 Classpath 或ServletContext等资源的handler，虽然可以注册自己的URLStreamHandler来解析特定的URL前缀（协议），比如“classpath:”，然而这需要了解URL的实现机制，而且URL也没有提供一些基本的方法，如检查当前资源是否存在、检查当前资源是否可读等方法。因而 Spring 对其内部使用到的资源实现了自己的抽象结构：Resource接口来封装底层资源。

```
public interface InputStreamSource {
    InputStream getInputStream() throws IOException;
}
public interface Resource extends InputStreamSource {
    boolean exists();
    boolean isReadable();
    boolean isOpen();
    URL getURL() throws IOException;
    URI getURI() throws IOException;
    File getFile() throws IOException;
    long lastModified() throws IOException;
    Resource createRelative(String relativePath) throws IOException;
    String getFilename();
    String getDescription();
}
```

InputStreamSource封装任何能返回InputStream的类，比如File、Classpath下的资源和Byte Array等。它只有一个方法定义：getInputStream()，该方法返回一个新的 InputStream对象。

Resource接口抽象了所有Spring内部使用到的底层资源：File、URL、Classpath等。首先，它定义了3个判断当前资源状态的方法：存在性（exists）、可读性（isReadable）、是否处于打开状态（isOpen）。另外，Resource接口还提供了不同资源到URL、URI、File类型的转换，以及获取lastModified属性、文件名（不带路径信息的文件名，getFilename()）的方法。为了便于操作，Resource 还提供了基于当前资源创建一个相对资源的方法：createRelative()。在错误处理中需要详细地打印出错的资源文件，因而Resource还提供了getDescription()方法用于在错误处理中的打印信息。

对不同来源的资源文件都有相应的Resource实现：文件（FileSystemResource）、Classpath资源（ClassPathResource）、URL资源（UrlResource）、InputStream资源（InputStreamResource）、Byte数组（ByteArrayResource）等。相关类图如2-8所示。

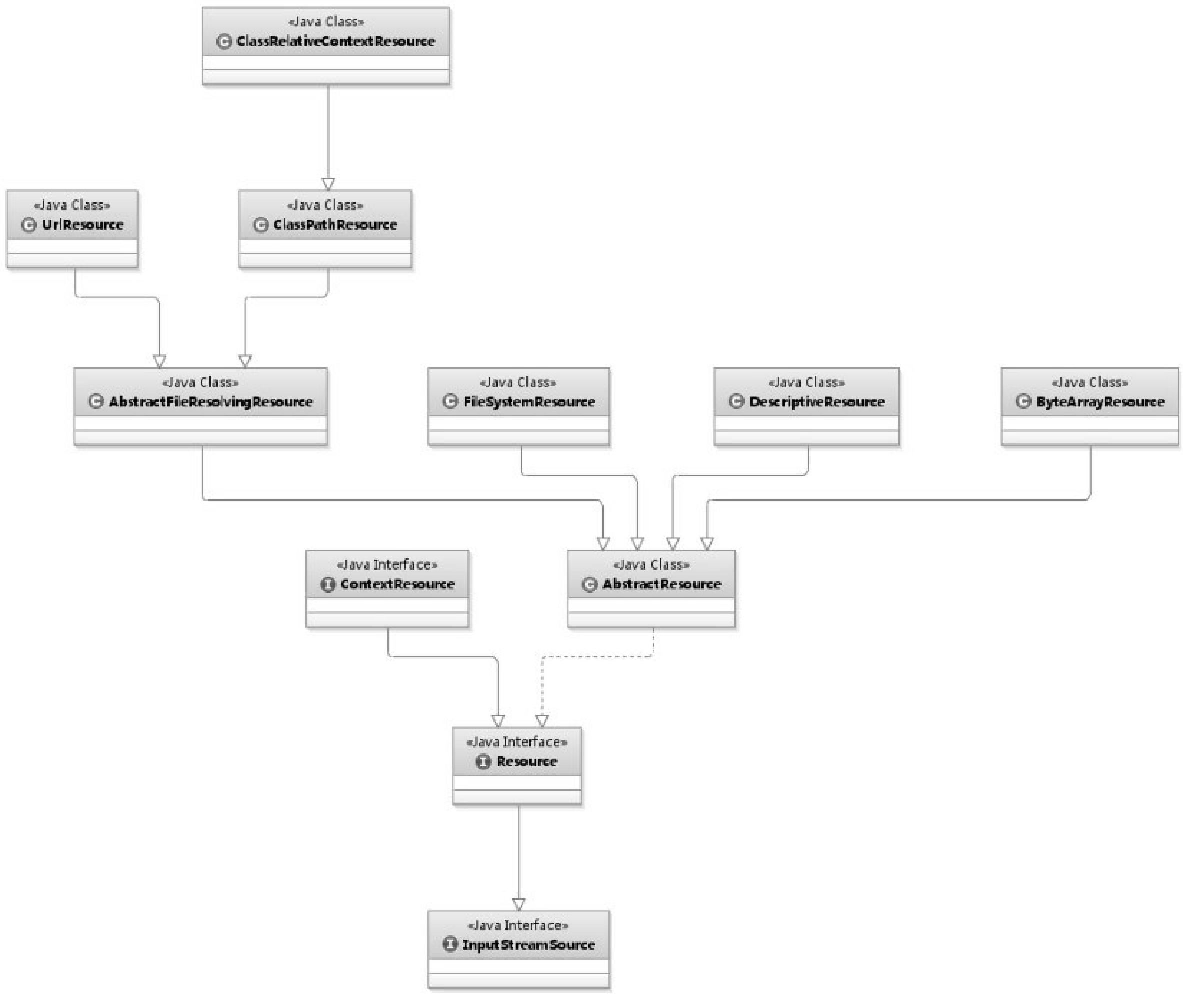


图2-8 资源文件处理相关类图

在日常的开发工作中，资源文件的加载也是经常用到的，可以直接使用Spring提供的类，比如在希望加载文件时可以使用以下代码：

Resource resource=new ClassPathResource(“beanFactoryTest.xml”);

InputStream inputStream=resource.getInputStream();

得到 inputStream 后，我们就可以按照以前的开发方式进行实现了，并且我们已经可以利用Resource及其子类为我们提供好的诸多特性。

有了Resource接口便可以对所有资源文件进行统一处理。至于实

现，其实是非常简单的，以getInputStream为例，ClassPathResource中的实现方式便是通过class或者classLoader提供的底层方法进行调用，而对于 FileSystemResource 的实现其实更简单，直接使用 FileInputStream对文件进行实例化。

ClassPathResource.java

```
if (this.clazz != null) {
    is = this.clazz.getResourceAsStream(this.path);
}else {
    is = this.classLoader.getResourceAsStream(this.path);
}
```

FileSystemResource.java

```
public InputStream getInputStream() throws IOException {
    return new FileInputStream(this.file);
}
```

当通过Resource相关类完成了对配置文件进行封装后配置文件的读取工作就全权交给XmlBeanDefinitionReader 来处理了。

了解了 Spring 中将配置文件封装为 Resource 类型的实例方法后，我们就可以继续探寻XmlBeanFactory的初始化过程了，XmlBeanFactory的初始化有若干办法，Spring中提供了很多的构造函数，在这里分析的是使用Resource实例作为构造函数参数的办法，代码如下：

XmlBeanFactory.java

```
public XmlBeanFactory(Resource resource) throws BeansException {
   //调用XmlBeanFactory（Resource,BeanFactory）构造方法，
      this(resource, null);
}
```

构造函数内部再次调用内部构造函数：

```
//parentBeanFactory为父类BeanFactory用于factory合并，可以为空
```

public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws

BeansException {

super(parentBeanFactory);

this.reader.loadBeanDefinitions(resource);

}

上面函数中的代码this.reader.loadBeanDefinitions(resource) 才是资源加载的真正实现，也是我们分析的重点之一。我们可以看到时序图中提到的XmlBeanDefinitionReader加载数据就是在这里完成的，但是在XmlBeanDefinitionReader加载数据前还有一个调用父类构造函数初始化的过程：super(parentBeanFactory)，跟踪代码到父类AbstractAutowireCapableBeanFactory的构造函数中：

AbstractAutowireCapableBeanFactory.java

public AbstractAutowireCapableBeanFactory() {

super();

ignoreDependencyInterface(BeanNameAware.class);

ignoreDependencyInterface(BeanFactoryAware.class);

ignoreDependencyInterface(BeanClassLoaderAware.class);

}

这里有必要提及一下ignoreDependencyInterface方法。ignoreDependencyInterface的主要功能是忽略给定接口的自动装配功能，那么，这样做的目的是什么呢？会产生什么样的效果呢？

举例来说，当A中有属性B，那么当Spring在获取A的Bean的时候如果其属性B还没有初始化，那么Spring会自动初始化B，这也是Spring中提供的一个重要特性。但是，某些情况下，B不会被初始化，其中的一种情况就是B实现了BeanNameAware接口。Spring中是这样介绍的：自动装配时忽略给定的依赖接口，典型应用是通过其他方式解析

Application上下文注册依赖，类似于 BeanFactory 通过 BeanFactoryAware 进行注入或者 ApplicationContext 通过 ApplicationContextAware进行注入。

2.5.2 加载Bean

之前提到的在XmlBeanFactory构造函数中调用了XmlBeanDefinitionReader类型的reader属性提供的方法this.reader.loadBeanDefinitions(resource)，而这句代码则是整个资源加载的切入点，我们先来看看这个方法的时序图，如图2-9所示。

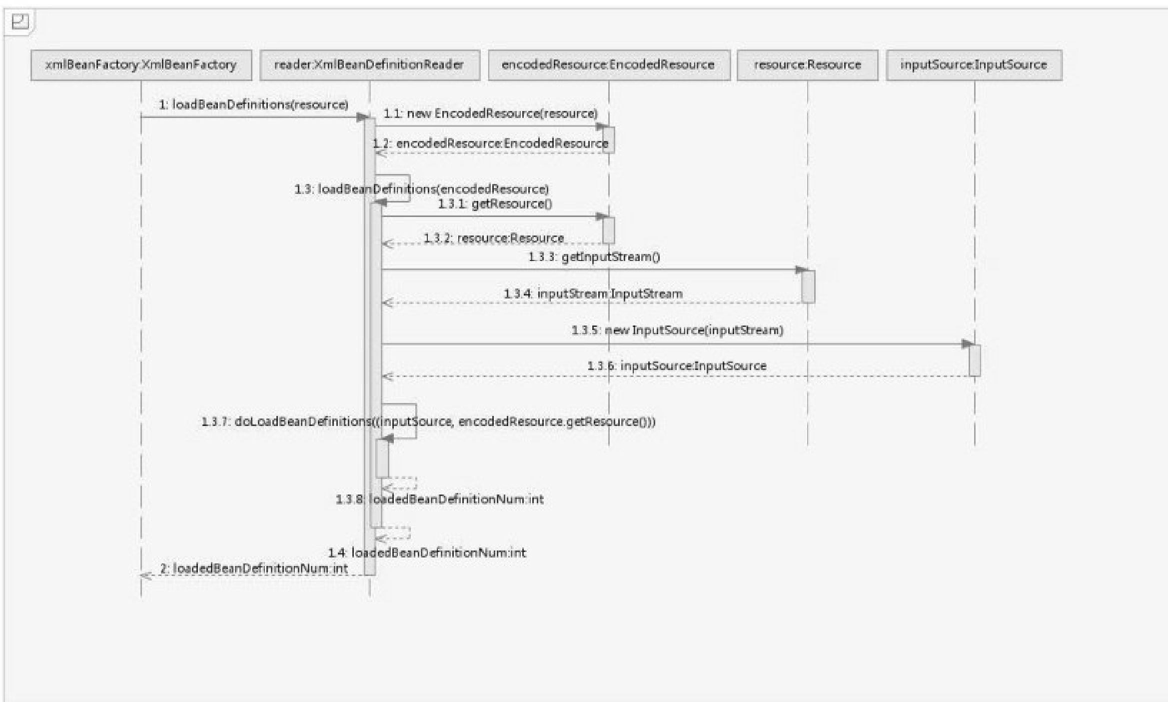


图2-9 loadBeanDefinitions函数执行时序图

看到图2-9我们才知道什么叫山路十八弯，绕了这么半天还没有真正地切入正题，比如加载XML文档和解析注册Bean，一直还在做准备工作。我们根据上面的时序图来分析一下这里究竟在准备什么？从上面的时序图中我们尝试梳理整个的处理过程如下。

（1）封装资源文件。当进入 XmlBeanDefinitionReader 后首先对参数 Resource 使用EncodedResource类进行封装。

（2）获取输入流。从Resource中获取对应的InputStream并构造InputSource。

（3）通过构造的InputSource实例和Resource实例继续调用函数doLoadBeanDefinitions。

我们来看一下loadBeanDefinitions函数具体的实现过程：

```
public int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException {
    return loadBeanDefinitions(new EncodedResource(resource));
}
```

那么EncodedResource的作用是什么呢？通过名称，我们可以大致推断这个类主要是用于对资源文件的编码进行处理的。其中的主要逻辑体现在getReader()方法中，当设置了编码属性的时候Spring会使用相应的编码作为输入流的编码。

```
public Reader getReader() throws IOException {
    if (this.encoding != null) {
        return new InputStreamReader(this.resource.getInputStream(),
this.encoding);
    }
    else {
        return new InputStreamReader(this.resource.getInputStream());
    }
}
```

上面代码构造了一个有编码（encoding）的InputStreamReader。当构造好encodedResource对象后，再次转入了可复用方法loadBeanDefinitions(new EncodedResource(resource))。

这个方法内部才是真正的数据准备阶段，也就是时序图所描述的逻辑：

```
public int loadBeanDefinitions(EncodedResource encodedResource)
throws BeanDefinitionStoreException {
    Assert.notNull(encodedResource, "EncodedResource must not be
null");
    if (logger.isInfoEnabled()) {
        logger.info("Loading XML bean definitions from " +
encodedResource. getResource());
    }
//通过属性来记录已经加载的资源
    Set<EncodedResource> currentResources =
this.resourcesCurrentlyBeingLoaded.get();
    if (currentResources == null) {
        currentResources = new HashSet<EncodedResource>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
            "Detected cyclic loading of " + encodedResource + " - check
your
            import definitions!");
    }
    try {
//从encodedResource中获取已经封装的Resource对象并再次从
Resource中获取其中的inputStream
    InputStream inputStream =
```

```
encodedResource.getResource().getInputStream();
            try {
            //InputSource这个类并不来自于Spring，它的全路径是
org.xml.sax.InputSource
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
               inputSource.setEncoding(encodedResource.getEncoding());
            }
            //真正进入了逻辑核心部分
                returndoLoadBeanDefinitions(inputSource,
encodedResource.getResource());
                }
                finally {
                   //关闭输入流
                   inputStream.close();
                }
            }
            catch (IOException ex) {
                throw new BeanDefinitionStoreException(
                    "IOException parsing XML document from " +
encodedResource.getResource(), ex);
            }
            finally {
               currentResources.remove(encodedResource);
                if (currentResources.isEmpty()) {
                   this.resourcesCurrentlyBeingLoaded.remove();
                }
```

```
    }
}
```

我们再次整理一下数据准备阶段的逻辑，首先对传入的resource参数做封装，目的是考虑到Resource可能存在编码要求的情况，其次，通过SAX读取XML文件的方式来准备InputSource对象，最后将准备的数据通过参数传入真正的核心处理部分doLoadBeanDefinitions(inputSource, encodedResource.getResource())。

```
protected int doLoadBeanDefinitions(InputSource inputSource,
Resource resource)
        throws BeanDefinitionStoreException {
        try {
            int validationMode = getValidationModeForResource(resource);
            Document doc = this.documentLoader.loadDocument(
                inputSource, getEntityResolver(), this.errorHandler,
validationMode,
                isNamespaceAware());
            return registerBeanDefinitions(doc, resource);
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (SAXParseException ex) {
            throw new
XmlBeanDefinitionStoreException(resource.getDescription(),
            "Line " + ex.getLineNumber() + " in XML document from " +
resource
            + " is invalid", ex);
```

```
        }
        catch (SAXException ex) {
            throw new
XmlBeanDefinitionStoreException(resource.getDescription(),
            "XML document from " + resource + " is invalid", ex);
        }
        catch (ParserConfigurationException ex) {
            throw new
BeanDefinitionStoreException(resource.getDescription(),
            "Parser configuration exception parsing XML from " + resource,
ex);
        }
        catch (IOException ex) {
            throw new
BeanDefinitionStoreException(resource.getDescription(),
            "IOException parsing XML document from " + resource, ex);
        }
        catch (Throwable ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
            "Unexpected exception parsing XML document from " + resource,
ex);
        }
    }
```

在上面冗长的代码中假如不考虑异常类的代码，其实只做了三件事，这三件事的每一件都必不可少。

（1）获取对XML文件的验证模式。

（2）加载XML文件，并得到对应的Document。

（3）根据返回的Document注册Bean信息。

这3个步骤支撑着整个Spring容器部分的实现基础，尤其是第3步对配置文件的解析，逻辑非常的复杂，那么我们先从获取XML文件的验证模式开始讲起。

2.6 获取XML的验证模式

了解XML文件的读者都应该知道XML文件的验证模式保证了XML文件的正确性，而比较常用的验证模式有两种：DTD和XSD。它们之间什么区别呢？

2.6.1 DTD与XSD区别

DTD（Document Type Definition）即文档类型定义，是一种XML约束模式语言，是XML文件的验证机制，属于XML文件组成的一部分。DTD是一种保证XML文档格式正确的有效方法，可以通过比较XML文档和DTD文件来看文档是否符合规范，元素和标签使用是否正确。一个DTD 文档包含：元素的定义规则，元素间关系的定义规则，元素可使用的属性，可使用的实体或符号规则。

要使用DTD验证模式的时候需要在XML文件的头部声明，以下是在Spring中使用DTD声明方式的代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//Spring//DTD BEAN 2.0//EN"
"http://www.Springframework. org/dtd/
Spring-beans-2.0.dtd">
<beans>
... ...
</beans>
```

而以Spring为例，具体的Spring-beans-2.0.dtd部分如下：

```
<!ELEMENT beans (
    description?,
     (import | alias | bean)*
)>
<!ATTLIST beans default-lazy-init (true | false) "false">
<!ATTLIST beans default-merge (true | false) "false">
<!ATTLIST beans default-autowire (no | byName | byType | constructor | autodetect) "no">
<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-init-method CDATA #IMPLIED>
<!ATTLIST beans default-destroy-method CDATA #IMPLIED>
... ...
```

XML Schema语言就是XSD（XML Schemas Definition）。XML Schema描述了XML文档的结构。可以用一个指定的XML Schema来验证某个XML文档，以检查该XML文档是否符合其要求。文档设计者可以通过XML Schema指定一个XML文档所允许的结构和内容，并可据此检查一个XML文档是否是有效的。XML Schema本身是一个XML文档，它符合XML语法结构。可以用通用的XML解析器解析它。

在使用XML Schema文档对XML实例文档进行检验，除了要声明名称空间外（xmlns= http://www.Springframework.org/schema/beans），还必须指定该名称空间所对应的XML Schema文档的存储位置。通过schemaLocation属性来指定名称空间所对应的XML Schema文档的存储位置，它包含两个部分，一部分是名称空间的URI，另一部分就是该名称空间所标识的XML Schema文件位置或URL地址（xsi:schemaLocation="http://www.Springframework.org/schema/beans http://www. Springframework.org/schema/beans/Spring-beans.xsd）。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
       http://www.Springframework.org/schema/beans/Spring-
beans.xsd">
    ... ...
</beans>
```

Spring-beans-3.0.xsd部分代码如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.Springframework.org/schema/beans">
<xsd:import namespace="http://www.w3.org/XML/1998/namespace"/>
<xsd:annotation>
   <xsd:documentation><![CDATA[
... ...
   ]]></xsd:documentation>
</xsd:annotation>
<!-- base types -->
<xsd:complexType name="identifiedType" abstract="true">
   <xsd:annotation>
      <xsd:documentation><![CDATA[
The unique identifier for a bean. The scope of the identifier
is the enclosing bean factory.
      ]]></xsd:documentation>
```

```
    </xsd:annotation>
     <xsd:attribute name="id" type="xsd:ID">
        <xsd:annotation>
            <xsd:documentation><![CDATA[
The unique identifier for a bean.
                ]]></xsd:documentation>
            </xsd:annotation>
        </xsd:attribute>
    </xsd:complexType>
     ... ...
</xsd:schema>
```

我们只是简单地介绍一下XML文件的验证模式的相关知识，目的在于让读者对后续知识的理解能有连续性，如果对XML有兴趣的读者可以进一步查阅相关资料。

2.6.2 验证模式的读取

了解了DTD与XSD的区别后我们再去分析Spring中对于验证模式的提取就更容易理解了。通过之前的分析我们锁定了Spring通过getValidationModeForResource方法来获取对应资源的的验证模式。

```
protected int getValidationModeForResource(Resource resource) {
     int validationModeToUse = getValidationMode();
    //如果手动指定了验证模式则使用指定的验证模式
     if (validationModeToUse != VALIDATION_AUTO) {
         return validationModeToUse;
    }
    //如果未指定则使用自动检测
     int detectedMode = detectValidationMode(resource);
```

```
        if (detectedMode != VALIDATION_AUTO) {
            return detectedMode;
        }
        return VALIDATION_XSD;
    }
```

方法的实现其实还是很简单的，无非是如果设定了验证模式则使用设定的验证模式（可以通过对调用XmlBeanDefinitionReader中的setValidationMode方法进行设定），否则使用自动检测的方式。而自动检测验证模式的功能是在函数detectValidationMode方法中实现的，在detectValidationMode函数中又将自动检测验证模式的工作委托给了专门处理类XmlValidationMode Detector，调用了XmlValidationModeDetector的validationModeDetector方法，具体代码如下：

```
    protected int detectValidationMode(Resource resource) {
        if (resource.isOpen()) {
            throw new BeanDefinitionStoreException(
            "Passed-in Resource [" + resource + "] contains an open stream: "
+
            "cannot determine validation mode automatically. Either pass in
            a Resource " +
            "that is able to create fresh streams, or explicitly specify the
            validationMode " +
            "on your XmlBeanDefinitionReader instance.");
        }
        InputStream inputStream;
        try {
            inputStream = resource.getInputStream();
        }
```

```
            catch (IOException ex) {
                throw new BeanDefinitionStoreException(
                "Unable to determine validation mode for [" + resource + "]:
cannot
                open InputStream. " +
                "Did you attempt to load directly from a SAX InputSource without
                specifying the " +
                "validationMode on your XmlBeanDefinitionReader instance?",
ex);
            }
            try {
                return
this.validationModeDetector.detectValidationMode(inputStream);
            }
            catch (IOException ex) {
                throw new BeanDefinitionStoreException("Unable to determine
validation mode for [" +
                resource + "]: an error occurred whilst reading from the
                InputStream.", ex);
            }
        }
        XmlValidationModeDetector.java
        public int detectValidationMode(InputStream inputStream) throws
IOException {
            BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
            try {
```

```
            boolean isDtdValidated = false;
            String content;
            while ((content = reader.readLine()) != null) {
                content = consumeCommentTokens(content);
               //如果读取的行是空或者是注释则略过
                if (this.inComment || !StringUtils.hasText(content)) {
                    continue;
                }
                if (hasDoctype(content)) {
                    isDtdValidated = true;
                    break;
                }
               //读取到<开始符号，验证模式一定会在开始符号之前
                if (hasOpeningTag(content)) {
                    break;
                }
            }
            return (isDtdValidated ? VALIDATION_DTD :
VALIDATION_XSD);
        }
        catch (CharConversionException ex) {
            // Choked on some character encoding...
            // Leave the decision up to the caller.
            return VALIDATION_AUTO;
        }
        finally {
            reader.close();
```

```
    }
}
private boolean hasDoctype(String content) {
    return (content.indexOf(DOCTYPE) > -1);
}
```

只要我们理解了XSD与DTD的使用方法，理解上面的代码应该不会太难，Spring用来检测验证模式的办法就是判断是否包含DOCTYPE，如果包含就是DTD，否则就是XSD。

2.7 获取Document

经过了验证模式准备的步骤就可以进行Document加载了，同样XmlBeanFactoryReader类对于文档读取并没有亲力亲为，而是委托给了DocumentLoader去执行，这里的DocumentLoader是个接口，而真正调用的是DefaultDocumentLoader，解析代码如下：

DefaultDocumentLoader.java

```
public Document loadDocument(InputSource inputSource,
EntityResolver entityResolver,
    Exception {
    ErrorHandler errorHandler, int validationMode, boolean
namespaceAware) throws
    DocumentBuilderFactory factory =
createDocumentBuilderFactory(validationMode,
   namespaceAware)
    if (logger.isDebugEnabled()) {
        logger.debug("Using JAXP provider [" +
factory.getClass().getName() + "]");
```

```
        }
        DocumentBuilder builder = createDocumentBuilder(factory,
entityResolver,
       errorHandler);
        return builder.parse(inputSource);
    }
```

对于这部分代码其实并没有太多可以描述的，因为通过SAX解析XML文档的套路大致都差不多，Spring 在这里并没有什么特殊的地方，同样首先创建 DocumentBuilderFactory，再通过DocumentBuilderFactory创建DocumentBuilder，进而解析inputSource来返回Document对象。对此感兴趣的读者可以在网上获取更多的资料。这里有必要提及一下EntityResolver，对于参数entityResolver，传入的是通过getEntityResolver()函数获取的返回值，如下代码：

```
    protected EntityResolver getEntityResolver() {
        if (this.entityResolver == null) {
            // Determine default EntityResolver to use.
            ResourceLoader resourceLoader = getResourceLoader();
            if (resourceLoader != null) {
                this.entityResolver = new
ResourceEntityResolver(resourceLoader);
            }
            else {
                this.entityResolver = new DelegatingEntityResolver
(getBeanClassLoader());
            }
        }
        return this.entityResolver;
```

}

那么，EntityResolver到底是做什么用的呢？

2.7.1 EntityResolver用法

在loadDocument方法中涉及一个参数EntityResolver，何为EntityResolver？官网这样解释:如果SAX应用程序需要实现自定义处理外部实体，则必须实现此接口并使用setEntityResolver方法向SAX 驱动器注册一个实例。也就是说，对于解析一个XML，SAX首先读取该XML文档上的声明，根据声明去寻找相应的DTD定义，以便对文档进行一个验证。默认的寻找规则，即通过网络（实现上就是声明的DTD的URI地址）来下载相应的DTD声明，并进行认证。下载的过程是一个漫长的过程，而且当网络中断或不可用时，这里会报错，就是因为相应的DTD声明没有被找到的原因。

EntityResolver的作用是项目本身就可以提供一个如何寻找DTD声明的方法，即由程序来实现寻找DTD声明的过程，比如我们将DTD文件放到项目中某处，在实现时直接将此文档读取并返回给SAX即可。这样就避免了通过网络来寻找相应的声明。

首先看entityResolver的接口方法声明：

InputSource resolveEntity(String publicId, String systemId)

这里，它接收两个参数publicId和systemId，并返回一个inputSource对象。这里我们以特定配置文件来进行讲解。

（1）如果我们在解析验证模式为XSD的配置文件，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
```

http://www.Springframework.org/schema/beans/Spring-beans.xsd">

... ...

</beans>

读取到以下两个参数。

publicId：null

systemId：http://www.Springframework.org/schema/beans/Spring-beans.xsd

（2）如果我们在解析验证模式为DTD的配置文件，代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC "-//Spring//DTD BEAN 2.0//EN" "http://www.Springframework.

org/dtd/Spring-beans-2.0.dtd">

<beans>

... ...

</beans>

读取到以下两个参数。

publicId：-//Spring//DTD BEAN 2.0//EN

systemId：http://www.Springframework.org/dtd/Spring-beans-2.0.dtd

之前已经提到过，验证文件默认的加载方式是通过 URL 进行网络下载获取，这样会造成延迟，用户体验也不好，一般的做法都是将验证文件放置在自己的工程里，那么怎么做才能将这个URL转换为自己工程里对应的地址文件呢？我们以加载DTD文件为例来看看Spring中是如何实现的。根据之前Spring中通过getEntityResolver()方法对EntityResolver的获取，我们知道，Spring中使用DelegatingEntityResolver类为EntityResolver的实现类，resolveEntity实现方法如下：

DelegatingEntityResolver.java

```
public InputSource resolveEntity(String publicId, String systemId)
throws
    SAXException, IOException {
        if (systemId != null) {
            if (systemId.endsWith(DTD_SUFFIX)) {
            //如果是dtd从这里解析
                return this.dtdResolver.resolveEntity(publicId, systemId);
            }
            else if (systemId.endsWith(XSD_SUFFIX)) {
              //通过调用META-INF/Spring.schemas解析
                return this.schemaResolver.resolveEntity(publicId, systemId);
            }
        }
        return null;
    }
```

我们可以看到，对不同的验证模式，Spring使用了不同的解析器解析。这里简单描述一下原理，比如加载DTD类型的BeansDtdResolver的resolveEntity是直接截取systemId最后的xx.dtd然后去当前路径下寻找，而加载XSD类型的PluggableSchemaResolver类的resolveEntity是默认到META-INF/Spring.schemas文件中找到systemid所对应的XSD文件并加载。

BeansDtdResolver.java

```
public InputSource resolveEntity(String publicId, String systemId)
throws IOException {
        if (logger.isTraceEnabled()) {
            logger.trace("Trying to resolve XML entity with public ID [" +
publicId +
```

```
        "] and system ID [" + systemId + "]");
    }
    // DTD_EXTENSION = ".dtd";
    if (systemId != null && systemId.endsWith(DTD_EXTENSION)) {
        int lastPathSeparator = systemId.lastIndexOf("/");
        for (String DTD_NAME : DTD_NAMES) {
            // DTD_NAMES = {"Spring-beans-2.0", "Spring-beans"};
            int dtdNameStart = systemId.indexOf(DTD_NAME);
            if (dtdNameStart > lastPathSeparator) {
                String dtdFile = systemId.substring(dtdNameStart);
                if (logger.isTraceEnabled()) {
                    logger.trace("Trying to locate [" + dtdFile + "] in Spring
jar");
                }
                try {
                    Resource resource = new ClassPathResource(dtdFile,
getClass());
                    InputSource source = new
InputSource(resource.getInputStream());
                    source.setPublicId(publicId);
                    source.setSystemId(systemId);
                    if (logger.isDebugEnabled()) {
                        logger.debug("Found beans DTD [" + systemId + "] in
                        classpath: " + dtdFile);
                    }
                    return source;
                }
```

```
                catch (IOException ex) {
                    if (logger.isDebugEnabled()) {
                        logger.debug("Could not resolve beans DTD [" +
systemId
                        + "]: not found in class path", ex);
                    }
                }
            }
        }
    }
    return null;
}
```

2.8 解析及注册BeanDefinitions

当把文件转换为Document后，接下来的提取及注册bean就是我们的重头戏。继续上面的分析，当程序已经拥有XML文档文件的Document实例对象时，就会被引入下面这个方法。

XmlBeanDefinitionReader.java

```
public int registerBeanDefinitions(Document doc, Resource resource)
throws BeanDefinitionStore
Exception {
    //使用DefaultBeanDefinitionDocumentReader实例化
BeanDefinitionDocumentReader
     BeanDefinitionDocumentReader documentReader =
createBeanDefinitionDocumentReader();
    //将环境变量设置其中
```

documentReader.setEnvironment(this.getEnvironment());
//在实例化 BeanDefinitionReader 时候会将 BeanDefinitionRegistry 传入，默认使用继承自
DefaultListableBeanFactory的子类
//记录统计前BeanDefinition的加载个数
int countBefore = getRegistry().getBeanDefinitionCount();
//加载及注册bean
documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
//记录本次加载的BeanDefinition个数
return getRegistry().getBeanDefinitionCount() - countBefore;
}

其中的参数 doc 是通过上一节 loadDocument 加载转换出来的。在这个方法中很好地应用了面向对象中单一职责的原则，将逻辑处理委托给单一的类进行处理，而这个逻辑处理类就是 BeanDefinitionDocumentReader。 BeanDefinitionDocumentReader是一个接口，而实例化的工作是在createBeanDefinitionDocumentReader()中完成的，而通过此方法，BeanDefinitionDocumentReader真正的类型其实已经是DefaultBeanDefinitionDocumentReader了，进入 DefaultBeanDefinition Document Reader 后，发现这个方法的重要目的之一就是提取 root，以便于再次将 root 作为参数继续BeanDefinition的注册。

public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
this.readerContext = readerContext;
logger.debug("Loading bean definitions");
Element root = doc.getDocumentElement();

```
    doRegisterBeanDefinitions(root);
}
```

经过艰难险阻，磕磕绊绊，我们终于到了核心逻辑的底部doRegisterBeanDefinitions(root)，至少我们在这个方法中看到了希望。

如果说以前一直是XML加载解析的准备阶段，那么doRegisterBeanDefinitions算是真正地开始进行解析了，我们期待的核心部分真正开始了。

```
protected void doRegisterBeanDefinitions(Element root) {
    //处理profile属性
     String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
     if (StringUtils.hasText(profileSpec)) {
        Assert.state(this.environment != null, "environment property must not be null");
        String[] specifiedProfiles = StringUtils.tokenizeToStringArray(profileSpec,

BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITEF
        if (!this.environment.acceptsProfiles(specifiedProfiles)) {
           return;
        }
    }
    //专门处理解析
     BeanDefinitionParserDelegate parent = this.delegate;
     this.delegate = createHelper(readerContext, root, parent);
    //解析前处理，留给子类实现
    preProcessXml(root);
    parseBeanDefinitions(root, this.delegate);
```

```
    //解析后处理，留给子类实现
    postProcessXml(root);
     this.delegate = parent;
}
```

通过上面的代码我们看到了处理流程，首先是对profile的处理，然后开始进行解析，可是当我们跟进preProcessXml(root)或者postProcessXml(root)发现代码是空的，既然是空的写着还有什么用呢？就像面向对象设计方法学中常说的一句话，一个类要么是面向继承的设计的，要么就用final修饰。在DefaultBeanDefinitionDocumentReader中并没有用final修饰，所以它是面向继承而设计的。这两个方法正是为子类而设计的，如果读者有了解过设计模式，可以很快速地反映出这是模版方法模式，如果继承自DefaultBeanDefinitionDocumentReader的子类需要在Bean解析前后做一些处理的话，那么只需要重写这两个方法就可以了。

2.8.1 profile属性的使用

我们注意到在注册Bean的最开始是对PROFILE_ATTRIBUTE属性的解析，可能对于我们来说，profile属性并不是很常用。让我们先了解一下这个属性。

分析profile前我们先了解下profile的用法，官方示例代码片段如下：

```
<beans xmlns="http://www.Springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jdbc="http://www.
    Springframework.org/schema/jdbc"
    xmlns:jee="http://www.Springframework.org/schema/jee"
    xsi:schemaLocation="...">
```

```
        ... ...
    <beans profile="dev">
      ... ...
    </beans>
      <beans profile="production">
        ... ...
      </beans>
    </beans>
```

集成到Web环境中时，在web.xml中加入以下代码：

```
<context-param>
<param-name>Spring.profiles.active</param-name>
<param-value>dev</param-value>
</context-param>
```

有了这个特性我们就可以同时在配置文件中部署两套配置来适用于生产环境和开发环境，这样可以方便的进行切换开发、部署环境，最常用的就是更换不同的数据库。

了解了 profile 的使用再来分析代码会清晰得多，首先程序会获取 beans 节点是否定义了profile 属性，如果定义了则会需要到环境变量中去寻找，所以这里首先断言 environment 不可能为空，因为 profile 是可以同时指定多个的，需要程序对其拆分，并解析每个 profile 是都符合环境变量中所定义的，不定义则不会浪费性能去解析。

2.8.2 解析并注册BeanDefinition

处理了 profile 后就可以进行 XML 的读取了，跟踪代码进入 parseBeanDefinitions(root, this.delegate)。

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
```

```
    //对beans的处理
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                   //对bean的处理
                    parseDefaultElement(ele, delegate);
                }
                else {
                   //对bean的处理
                   delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
       delegate.parseCustomElement(root);
    }
}
```

上面的代码看起来逻辑还是蛮清晰的，因为在Spring的XML配置里面有两大类Bean声明，一个是默认的，如：

<bean id="test" class="test.TestBean"/>

另一类就是自定义的，如：

<tx:annotation-driven/>

而两种方式的读取及解析差别是非常大的，如果采用Spring默认的配置，Spring当然知道该怎么做，但是如果是自定义的，那么就需要用户实现一些接口及配置了。对于根节点或者子节点如果是默认命名空间的话则采用 parseDefaultElement 方法进行解析，否则使用 delegate.parseCustomElement方法对自定义命名空间进行解析。而判断是否默认命名空间还是自定义命名空间的办法其实是使用 node.getNamespaceURI()获取命名空间，并与Spring中固定的命名空间 http://www.Springframework.org/schema/beans 进行比对。如果一致则认为是默认，否则就认为是自定义。而对于默认标签解析与自定义标签解析我们将会在下一章中进行讨论。

第**3**章　默认标签的解析

之前提到过Spring中的标签包括默认标签和自定义标签两种，而两种标签的用法以及解析方式存在着很大的不同，本章节重点带领读者详细分析默认标签的解析过程。

默认标签的解析是在parseDefaultElement函数中进行的，函数中的功能逻辑一目了然，分别对4种不同标签（import、alias、bean和beans）做了不同的处理。

```
    private void parseDefaultElement(Element ele,
BeanDefinitionParserDelegate delegate) {
        //对import标签的处理
            if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
              importBeanDefinitionResource(ele);
            }
        //对alias标签的处理
            else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
```

```
            processAliasRegistration(ele);
        }
    //对bean标签的处理
        else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
            processBeanDefinition(ele, delegate);
        }
    //对beans标签的处理
        else if (delegate.nodeNameEquals(ele, 
NESTED_BEANS_ELEMENT)) {
            doRegisterBeanDefinitions(ele);
        }
    }
```

[3.1 bean标签的解析及注册]()

在 4 种标签的解析中，对 bean 标签的解析最为复杂也最为重要，所以我们从此标签开始深入分析，如果能理解此标签的解析过程，其他标签的解析自然会迎刃而解。首先我们进入函数 processBeanDefinition(ele, delegate)。

```
    protected void processBeanDefinition(Element ele, 
BeanDefinitionParserDelegate delegate) {
        BeanDefinitionHolder bdHolder = 
delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, 
bdHolder);
            try {
```

```
getRegistry());
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext(). }
            catch (BeanDefinitionStoreException ex) {
               getReaderContext().error("Failed to register bean definition with
name '" +
               bdHolder.getBeanName() + "'", ele, ex);
            }
            getReaderContext().fireComponentRegistered(new
BeanComponentDefinition(bdHolder));
         }
      }
```

乍一看，似乎一头雾水，没有以前的函数那样清晰的逻辑。大致的逻辑总结如下。

（1）首先委托BeanDefinitionDelegate类的parseBeanDefinitionElement方法进行元素解析，返回BeanDefinitionHolder类型的实例bdHolder，经过这个方法后，bdHolder实例已经包含我们配置文件中配置的各种属性了，例如class、name、id、alias之类的属性。

（2）当返回的 bdHolder 不为空的情况下若存在默认标签的子节点下再有自定义属性，还需要再次对自定义标签进行解析。

（3）解析完成后，需要对解析后的 bdHolder 进行注册，同样，注册操作委托给了 Bean DefinitionReaderUtils的registerBeanDefinition方法。

（4）最后发出响应事件，通知想关的监听器，这个bean已经加载完成了。

配合时序图（如图3-1所示），可能会更容易理解。

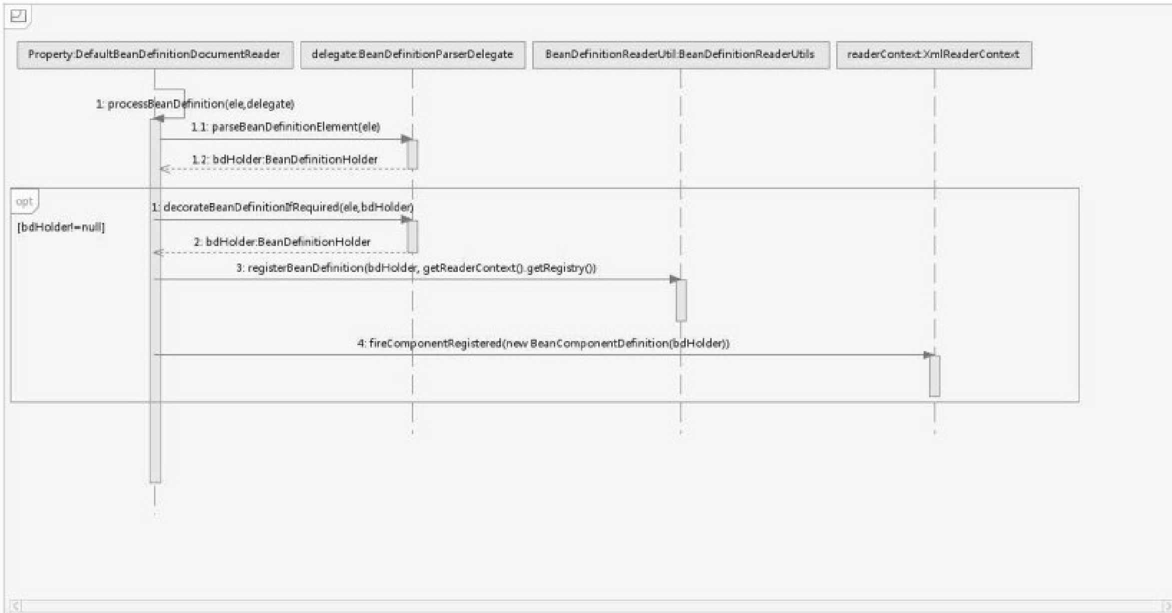


图3-1 bean标签的解析及注册时序图

3.1.1 解析BeanDefinition

下面我们就针对各个操作做具体分析。首先我们从元素解析及信息提取开始，也就是BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele)，进入BeanDefinition Delegate类的parseBeanDefinitionElement方法。

BeanDefinitionDelegate.java

```
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele)
{
    return parseBeanDefinitionElement(ele, null);
}
public BeanDefinitionHolder parseBeanDefinitionElement(Element ele,
BeanDefinition
containingBean) {
```

```
//解析id属性
    String id = ele.getAttribute(ID_ATTRIBUTE);
//解析name属性
 String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
//分割name属性
    List<String> aliases = new ArrayList<String>();
    if (StringUtils.hasLength(nameAttr)) {
        String[] nameArr =
StringUtils.tokenizeToStringArray(nameAttr, MULTI_
       VALUE_ATTRIBUTE_DELIMITERS);
       aliases.addAll(Arrays.asList(nameArr));
    }
    String beanName = id;
    if (!StringUtils.hasText(beanName) && !aliases.isEmpty()) {
        beanName = aliases.remove(0);
        if (logger.isDebugEnabled()) {
            logger.debug("No XML 'id' specified - using '" + beanName +
                "' as bean name and " + aliases + " as aliases");
        }
    }
    if (containingBean == null) {
        checkNameUniqueness(beanName, aliases, ele);
    }
    AbstractBeanDefinition beanDefinition =
parseBeanDefinitionElement(ele, beanName,
   containingBean);
    if (beanDefinition != null) {
```

```
                    if (!StringUtils.hasText(beanName)) {
                        try {
                            //如果不存在beanName那么根据Spring中提供的命名规
则为当前bean生成对应
                            的beanName
                            if (containingBean != null) {
                                beanName =
BeanDefinitionReaderUtils.generateBeanName(
                                beanDefinition, this.readerContext.getRegistry(), true);
                            }
                        else {
                            beanName =
this.readerContext.generateBeanName(beanDefinition);
                            String beanClassName =
beanDefinition.getBeanClassName();
                            if (beanClassName != null &&
                                > beanClassName.length() &&
                                beanName.startsWith(beanClassName) &&
beanName.length()
                                    (beanClassName)) {
                                    !this.readerContext.getRegistry(). IsBeanNameInUse
                                aliases.add(beanClassName);
                            }
                        }
                        if (logger.isDebugEnabled()) {
                            logger.debug("Neither XML 'id' nor 'name' specified - " +
                            "using generated bean name [" + beanName + "]");
```

```
                }
            }
            catch (Exception ex) {
                error(ex.getMessage(), ele);
                return null;
            }
        }
        String[] aliasesArray = StringUtils.toStringArray(aliases);
        return new BeanDefinitionHolder(beanDefinition, beanName, 
aliasesArray);
    }
    return null;
}
```

以上便是对默认标签解析的全过程了。当然，对Spring的解析犹如洋葱剥皮一样，一层一层地进行，尽管现在只能看到对属性id以及name的解析，但是很庆幸，思路我们已经了解了。在开始对属性展开全面解析前，Spring在外层又做了一个当前层的功能架构，在当前层完成的主要工作包括如下内容。

（1）提取元素中的id以及name属性。

（2）进一步解析其他所有属性并统一封装至GenericBeanDefinition类型的实例中。

（3）如果检测到bean没有指定beanName，那么使用默认规则为此Bean生成beanName。

（4）将获取到的信息封装到BeanDefinitionHolder的实例中。

我们进一步地查看步骤（2）中对标签其他属性的解析过程。

```
public AbstractBeanDefinition parseBeanDefinitionElement(
Element ele, String beanName, BeanDefinition containingBean) {
```

```
this.parseState.push(new BeanEntry(beanName));
String className = null;
//解析class属性
if (ele.hasAttribute(CLASS_ATTRIBUTE)) {
    className = ele.getAttribute(CLASS_ATTRIBUTE).trim();
}
try {
    String parent = null;
    //解析parent属性
    if (ele.hasAttribute(PARENT_ATTRIBUTE)) {
        parent = ele.getAttribute(PARENT_ATTRIBUTE);
    }
    //创建用于承载属性的AbstractBeanDefinition类型的
GenericBeanDefinition
    AbstractBeanDefinition bd = createBeanDefinition(className,
parent);
    //硬编码解析默认bean的各种属性
    parseBeanDefinitionAttributes(ele, beanName, containingBean,
bd);
    //提取description
    ELEMENT));

bd.setDescription(DomUtils.getChildElementValueByTagName(ele,
DESCRIPTION_
    //解析元数据
    parseMetaElements(ele, bd);
    //解析lookup-method属性
```

```
            parseLookupOverrideSubElements(ele, bd.getMethodOverrides());
            //解析replaced-method属性
            parseReplacedMethodSubElements(ele, bd.getMethodOverrides());
            //解析构造函数参数
            parseConstructorArgElements(ele, bd);
            //解析property子元素
            parsePropertyElements(ele, bd);
            //解析qualifier子元素
            parseQualifierElements(ele, bd);
            bd.setResource(this.readerContext.getResource());
            bd.setSource(extractSource(ele));
             return bd;
        }
         catch (ClassNotFoundException ex) {
             error("Bean class [" + className + "] not found", ele, ex);
        }
         catch (NoClassDefFoundError err) {
             error("Class that bean class [" + className + "] depends on not
found", ele, err);
        }
         catch (Throwable ex) {
             error("Unexpected failure during bean definition parsing", ele, ex);
        }
         finally {
            this.parseState.pop();
        }
         return null;
```

}

终于，bean标签的所有属性，不论常用的还是不常用的我们都看到了，尽管有些复杂的属性还需要进一步的解析，不过丝毫不会影响我们兴奋的心情。接下来，我们继续一些复杂标签属性的解析。

1．创建用于属性承载的**BeanDefinition**

BeanDefinition 是一个接口，在 Spring 中存在三种实现：RootBeanDefinition、ChildBean Definition 以及 GenericBeanDefinition。三种实现均继承了 AbstractBeanDefiniton，其中BeanDefinition 是配置文件<bean>元素标签在容器中的内部表示形式。<bean>元素标签拥有class、scope、lazy-init等配置属性，BeanDefinition则提供了相应的beanClass、scope、lazyInit属性，BeanDefinition和<bean>中的属性是一一对应的。其中RootBeanDefinition是最常用的实现类，它对应一般性的<bean>元素标签，GenericBeanDefinition是自2.5版本以后新加入的bean文件配置属性定义类，是一站式服务类。

在配置文件中可以定义父<bean>和子<bean>，父<bean>用RootBeanDefinition表示，而子<bean>用ChildBeanDefiniton表示，而没有父<bean>的<bean>就使用RootBeanDefinition表示。AbstractBeanDefinition对两者共同的类信息进行抽象。

Spring通过BeanDefinition将配置文件中的<bean>配置信息转换为容器的内部表示，并将这些BeanDefiniton注册到BeanDefinitonRegistry中。Spring容器的BeanDefinitionRegistry就像是Spring配置信息的内存数据库，主要是以map的形式保存，后续操作直接从BeanDefinition Registry中读取配置信息。它们之间的关系如图3-2所示。

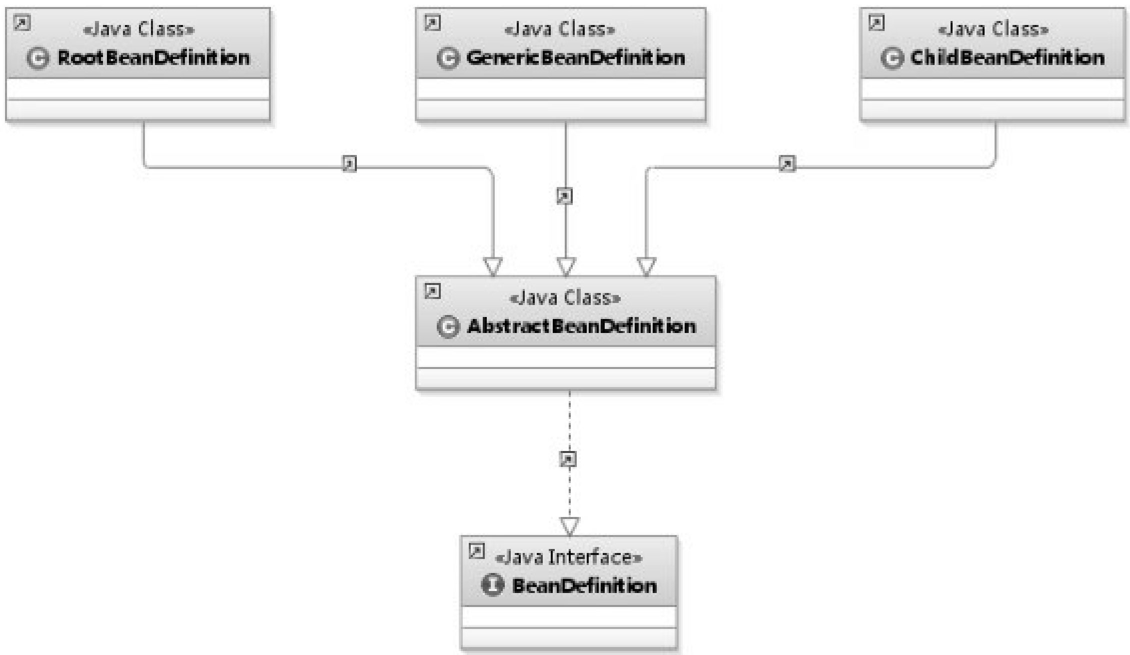


图3-2 BeanDefinition及其实现类

由此可知，要解析属性首先要创建用于承载属性的实例，也就是创建GenericBeanDefinition类型的实例。而代码createBeanDefinition(className, parent)的作用就是实现此功能。

```
protected AbstractBeanDefinition createBeanDefinition(String className, String parentName)
    throws ClassNotFoundException {
        return BeanDefinitionReaderUtils.createBeanDefinition(
        parentName, className, this.readerContext.getBeanClassLoader());
    }
BeanDefinitionReaderUtils.java
public static AbstractBeanDefinition createBeanDefinition(
String parentName, String className, ClassLoader classLoader) throws
ClassNotFoundException {
    GenericBeanDefinition bd = new GenericBeanDefinition();
```

//parentName可能为空

bd.setParentName(parentName);

if (className != null) {

if (classLoader != null) {

//如果classLoader不为空，则使用以传入的classLoader同一虚拟机加载类对象，否则只是

记录className

bd.setBeanClass(ClassUtils.forName(className, classLoader));

}

else {

bd.setBeanClassName(className);

}

}

return bd;

}

2．解析各种属性

当我们创建了bean信息的承载实例后，便可以进行bean信息的各种属性解析了，首先我们进入parseBeanDefinitionAttributes方法。parseBeanDefinitionAttributes方法是对element所有元素属性进行解析：

public AbstractBeanDefinition parseBeanDefinitionAttributes(Element ele, String beanName,

BeanDefinition containingBean, AbstractBeanDefinition bd) {

//解析scope属性

if (ele.hasAttribute(SCOPE_ATTRIBUTE)) {

// Spring 2.x "scope" attribute

bd.setScope(ele.getAttribute(SCOPE_ATTRIBUTE));

if (ele.hasAttribute(SINGLETON_ATTRIBUTE)) {

```
            //scope与singleton两个属性只能指定其中之一，不可以同时出
现，否则Spring将会报出异常
                error("Specify either 'scope' or 'singleton', not both", ele);
               }
            }
            //解析singleton属性
             else if (ele.hasAttribute(SINGLETON_ATTRIBUTE)) {
                // Spring 1.x "singleton" attribute

bd.setScope(TRUE_VALUE.equals(ele.getAttribute(SINGLETON_ATTRIBU
?
                    BeanDefinition.SCOPE_SINGLETON :
BeanDefinition.SCOPE_PROTOTYPE);
            }
             else if (containingBean != null) {
             // Take default from containing bean in case of an inner bean
definition.
            //在嵌入beanDifinition情况下且没有单独指定scope属性则使用父
类默认的属性
               bd.setScope(containingBean.getScope());
            }
            //解析abstract属性
        if (ele.hasAttribute(ABSTRACT_ATTRIBUTE)) {

bd.setAbstract(TRUE_VALUE.equals(ele.getAttribute(ABSTRACT_ATTRIB
        }
        //解析lazy-init属性
```

```
        String lazyInit = ele.getAttribute(LAZY_INIT_ATTRIBUTE);
        if (DEFAULT_VALUE.equals(lazyInit)) {
            lazyInit = this.defaults.getLazyInit();
        }
        //若没有设置或设置成其他字符都会被设置为false
        bd.setLazyInit(TRUE_VALUE.equals(lazyInit));
        //解析autowire属性
        String autowire = ele.getAttribute(AUTOWIRE_ATTRIBUTE);
        bd.setAutowireMode(getAutowireMode(autowire));
        //解析dependency-check属性
        String dependencyCheck =
ele.getAttribute(DEPENDENCY_CHECK_ATTRIBUTE);
        bd.setDependencyCheck(getDependencyCheck(dependencyCheck));
        //解析depends-on属性
        if (ele.hasAttribute(DEPENDS_ON_ATTRIBUTE)) {
            String dependsOn = ele.getAttribute(DEPENDS_ON_ATTRIBUTE);
            bd.setDependsOn(StringUtils.tokenizeToStringArray(dependsOn,
MULTI_VALUE_
           ATTRIBUTE_DELIMITERS));
        }
        //解析autowire-candidate属性
        String autowireCandidate =
ele.getAttribute(AUTOWIRE_CANDIDATE_ATTRIBUTE);
        if ("".equals(autowireCandidate) ||
DEFAULT_VALUE.equals(autowireCandidate)) {
            String candidatePattern = this.defaults.getAutowireCandidates();
            if (candidatePattern != null) {
```

```
            String[] patterns = StringUtils.commaDelimitedListToStringArray
            (candidatePattern);

bd.setAutowireCandidate(PatternMatchUtils.simpleMatch(patterns,
beanName));
        }
    }
    else {

bd.setAutowireCandidate(TRUE_VALUE.equals(autowireCandidate));
    }
    //解析primary属性
    if (ele.hasAttribute(PRIMARY_ATTRIBUTE)) {

bd.setPrimary(TRUE_VALUE.equals(ele.getAttribute(PRIMARY_ATTRIBU
    }
    //解析init-method属性
    if (ele.hasAttribute(INIT_METHOD_ATTRIBUTE)) {
        String initMethodName =
ele.getAttribute(INIT_METHOD_ATTRIBUTE);
        if (!"".equals(initMethodName)) {
          bd.setInitMethodName(initMethodName);
        }
    }
    else {
        if (this.defaults.getInitMethod() != null) {
          bd.setInitMethodName(this.defaults.getInitMethod());
```

```
            bd.setEnforceInitMethod(false);
        }
    }
    //解析destroy-method属性
    if (ele.hasAttribute(DESTROY_METHOD_ATTRIBUTE)) {
        String destroyMethodName =
ele.getAttribute(DESTROY_METHOD_ATTRIBUTE);
        if (!"".equals(destroyMethodName)) {
            bd.setDestroyMethodName(destroyMethodName);
        }
    }
    else {
        if (this.defaults.getDestroyMethod() != null) {
            bd.setDestroyMethodName(this.defaults.getDestroyMethod());
            bd.setEnforceDestroyMethod(false);
        }
    }
    //解析factory-method属性
    if (ele.hasAttribute(FACTORY_METHOD_ATTRIBUTE)) {

bd.setFactoryMethodName(ele.getAttribute(FACTORY_METHOD_ATTRIB
    }
    //解析factory-bean属性
    if (ele.hasAttribute(FACTORY_BEAN_ATTRIBUTE)) {

bd.setFactoryBeanName(ele.getAttribute(FACTORY_BEAN_ATTRIBUTE)
    }
```

```
        return bd;
    }
```

我们可以清楚地看到Spring完成了对所有bean属性的解析，这些属性中有很多是我们经常使用的，同时我相信也一定会有或多或少的属性是读者不熟悉或者是没有使用过的，有兴趣的读者可以查阅相关资料进一步了解每个属性。

3．解析子元素**meta**

在开始解析元数据的分析前，我们先回顾下元数据meta属性的使用。

```
<bean id="myTestBean" class="bean.MyTestBean">
    <meta key="testStr" value="aaaaaaaa"/>
</bean>
```

这段代码并不会体现在 MyTestBean 的属性当中，而是一个额外的声明，当需要使用里面的信息的时候可以通过BeanDefinition的getAttribute(key)方法进行获取。

对meta属性的解析代码如下：

```
public void parseMetaElements(Element ele, BeanMetadataAttributeAccessor attributeAccessor) {
    //获取当前节点的所有子元素
    NodeList nl = ele.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        //提取meta
        if (isCandidateElement(node) && nodeNameEquals(node, META_ELEMENT)) {
            Element metaElement = (Element) node;
            String key = metaElement.getAttribute(KEY_ATTRIBUTE);
```

```
            String value = 
metaElement.getAttribute(VALUE_ATTRIBUTE);
            //使用key、value构造BeanMetadataAttribute
            BeanMetadataAttribute attribute = new 
BeanMetadataAttribute(key, value);
            attribute.setSource(extractSource(metaElement));
            //记录信息
            attributeAccessor.addMetadataAttribute(attribute);
        }
    }
}
```

4．解析子元素lookup-method

同样，子元素 lookup-method 似乎并不是很常用，但是在某些时候它的确是非常有用的属性，通常我们称它为获取器注入。引用《Spring in Action》中的一句话：获取器注入是一种特殊的方法注入，它是把一个方法声明为返回某种类型的 bean，但实际要返回的 bean 是在配置文件里面配置的，此方法可用在设计有些可插拔的功能上，解除程序依赖。我们看看具体的应用。

（1）首先我们创建一个父类。

```
package test.lookup.bean;
public class User {
    public void showMe(){
        System.out.println("i am user");
    }
}
```

（2）创建其子类并覆盖showMe方法。

```
package test.lookup.bean;
```

```
public class Teacher extends User{
    public void showMe(){
        System.out.println("i am Teacher");
    }
}
```

（3）创建调用方法。

```
public abstract class GetBeanTest {
    public void showMe(){
        this.getBean().showMe();
    }
    public abstract User getBean();
}
```

（4）创建测试方法。

```
package test.lookup;
import org.Springframework.context.ApplicationContext;
import
org.Springframework.context.support.ClassPathXmlApplicationContext;
import test.lookup.app.GetBeanTest;
public class Main {
    public static void main(String[] args) {
        ApplicationContext bf =
            new
ClassPathXmlApplicationContext("test/lookup/lookupTest.xml");
        GetBeanTest test=(GetBeanTest) bf.getBean("getBeanTest");
        test.showMe();
    }
}
```

到现在为止，除了配置文件外，整个测试方法就完成了，如果之前没有接触过获取器注入的读者们可能会有疑问：抽象方法还没有被实现，怎么可以直接调用呢？答案就在Spring为我们提供的获取器中，我们看看配置文件是怎么配置的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www. Springframework.
   org/schema/beans/Spring-beans.xsd">
      <bean id="getBeanTest" class="test.lookup.app.GetBeanTest">
      <lookup-method name="getBean" bean="teacher"/>
   </bean>
<bean id="teacher" class="test.lookup.bean.Teacher"/>
</beans>
```

在配置文件中，我们看到了源码解析中提到的lookup-method子元素，这个配置完成的功能是动态地将teacher所代表的bean作为getBean的返回值，运行测试方法我们会看到控制台上的输出：

```
i am Teacher
```

当我们的业务变更或者在其他情况下，teacher 里面的业务逻辑已经不再符合我们的业务要求，需要进行替换怎么办呢？这是我们需要增加新的逻辑类：

```
package test.lookup.bean;
public class Student extends User {
    public void showMe(){
        System.out.println("i am student");
```

```
    }
}
```

同时修改配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www.Springframework.
   org/schema/beans/Spring-beans.xsd">
      <bean id="getBeanTest" class="test.lookup.app.GetBeanTest">
     <lookup-method name="getBean" bean="student"/>
   </bean>
<bean id="teacher" class="test.lookup.bean.Teacher"/>
<bean id="student" class="test.lookup.bean.Student"/>
</beans>
```

再次运行测试类，你会发现不一样的结果：

```
i am Student
```

至此，我们已经初步了解了 lookup-method 子元素所提供的大致功能，相信这时再次去看它的属性提取源码会觉得更有针对性。

```
public void parseLookupOverrideSubElements(Element beanEle,
MethodOverrides overrides) {
     NodeList nl = beanEle.getChildNodes();
     for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        //仅当在Spring默认bean的子元素下且为 <lookup-method时有
效
```

```
                METHOD_ELEMENT)) {
                 if (isCandidateElement(node) && nodeNameEquals(node,
LOOKUP_
                     Element ele = (Element) node;
                    //获取要修饰的方法
                     String methodName = ele.getAttribute(NAME_ATTRIBUTE);
                    //获取配置返回的bean
                     String beanRef = ele.getAttribute(BEAN_ELEMENT);
                     LookupOverride override = new LookupOverride(methodName,
beanRef);
                    override.setSource(extractSource(ele));
                    overrides.addOverride(override);
                }
            }
        }
```

上面的代码很眼熟，似乎与parseMetaElements的代码大同小异，最大的区别就是在if判断中的节点名称在这里被修改为LOOKUP_METHOD_ELEMENT。还有，在数据存储上面通过使用LookupOverride 类型的实体类来进行数据承载并记录在AbstractBeanDefinition 中的methodOverrides属性中。

5．解析子元素replaced-method

这个方法主要是对bean中replaced-method子元素的提取，在开始提取分析之前我们还是预先介绍下这个元素的用法。

方法替换：可以在运行时用新的方法替换现有的方法。与之前的look-up 不同的是， replaced-method不但可以动态地替换返回实体bean，而且还能动态地更改原有方法的逻辑。我们来看看使用示例。

（1）在changeMe中完成某个业务逻辑。

```
public class TestChangeMethod {
    public void changeMe(){
        System.out.println("changeMe");
    }
}
```

（2）在运营一段时间后需要改变原有的业务逻辑。

```
public class TestMethodReplacer implements MethodReplacer{
@Override
public Object reimplement(Object obj, Method method, Object[]
args)throws Throwable {
    System.out.println("我替换了原有的方法");
        return null;
    }
}
```

（3）使替换后的类生效。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www.Springframework.
org/schema/beans/Spring-beans.xsd">
<bean id="testChangeMethod"
class="test.replacemethod.TestChangeMethod">
    <replaced-method name="changeMe" replacer="replacer"/>
</bean>
<bean id="replacer" class="test.replacemethod.TestMethodReplacer"/>
</beans>
```

（4）测试。

```
public static void main(String[] args) {
    ApplicationContext bf =
      new
ClassPathXmlApplicationContext("test/replacemethod/replaceMethodTest.xm
    TestChangeMethod test=(TestChangeMethod)
bf.getBean("testChangeMethod");
    test.changeMe();
    }
```

好了，运行测试类就可以看到预期的结果了，控制台成功打印出“我替换了原有的方法”，也就是说我们做到了动态替换原有方法，知道了这个元素的用法，我们再次来看元素的提取过程：

```
public void parseReplacedMethodSubElements(Element beanEle,
MethodOverrides overrides) {
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
      Node node = nl.item(i);
      //仅当在Spring默认bean的子元素下且为<replaced-method时有
效
      ELEMENT)) {
      if (isCandidateElement(node) && nodeNameEquals(node,
REPLACED_METHOD_
        Element replacedMethodEle = (Element) node;
        //提取要替换的旧的 方法
        String name =
replacedMethodEle.getAttribute(NAME_ATTRIBUTE);
        //提取对应的新的替换方法
```

```
            String callback =
replacedMethodEle.getAttribute(REPLACER_ATTRIBUTE);
            ReplaceOverride replaceOverride = new
ReplaceOverride(name, callback);
            List<Element> argTypeEles =
DomUtils.getChildElementsByTagName
            (replacedMethodEle, ARG_TYPE_ELEMENT);
            for (Element argTypeEle : argTypeEles) {
              //记录参数
               String match = argTypeEle.getAttribute
(ARG_TYPE_MATCH_ATTRIBUTE);
               match = (StringUtils.hasText(match) ? match :
DomUtils.getTextValue
              (argTypeEle));
               if (StringUtils.hasText(match)) {
                 replaceOverride.addTypeIdentifier(match);
              }
           }
           replaceOverride.setSource(extractSource(replacedMethodEle));
           overrides.addOverride(replaceOverride);
        }
     }
   }
```

我们可以看到无论是look-up还是replaced-method都是构造了一个MethodOverride，并最终记录在了AbstractBeanDefinition中的methodOverrides属性中。而这个属性如何使用以完成它所提供的功能我们会在后续的章节进行详细地介绍。

6．解析子元素**constructor-arg**

对构造函数的解析是非常常用的，同时也是非常复杂的，也相信大家对构造函数的配置都不陌生，举个简单的小例子：

```
... ...
<beans>
<!-- 默认的情况下是按照参数的顺序注入，当指定index索引后就可以改变注入参数的顺序 -->
<bean id="helloBean" class="com.HelloBean">
     <constructor-arg index="0">
        <value>郝佳</value>
    </constructor-arg>
     <constructor-arg index="1">
        <value>你好</value>
    </constructor-arg>
</bean>
... ...
</beans>
```

上面的配置是Spring构造函数配置中最基础的配置，实现的功能就是对HelloBean自动寻找对应的构造函数，并在初始化的时候将设置的参数传入进去。那么让我们来看看具体的 XML解析过程。

对于constructor-arg子元素的解析，Spring是通过parseConstructorArgElements函数来实现的，具体的代码如下：

```
public void parseConstructorArgElements(Element beanEle, BeanDefinition bd) {
     NodeList nl = beanEle.getChildNodes();
     for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
```

```
                if (isCandidateElement(node) && nodeNameEquals(node,
CONSTRUCTOR_ARG_ELEMENT)) {
                    //解析constructor-arg
                    parseConstructorArgElement((Element) node, bd);
                }
            }
        }
```

这个结构似乎我们可以想象得到，遍历所有子元素，也就是提取所有constructor-arg，然后进行解析，但是具体的解析却被放置在了另个函数parseConstructorArgElement中，具体代码如下：

```
        public void parseConstructorArgElement(Element ele, BeanDefinition
bd) {
            //提取index属性
             String indexAttr = ele.getAttribute(INDEX_ATTRIBUTE);
            //提取type属性
             String typeAttr = ele.getAttribute(TYPE_ATTRIBUTE);
            //提取name属性
             String nameAttr = ele.getAttribute(NAME_ATTRIBUTE);
             if (StringUtils.hasLength(indexAttr)) {
                 try {
                     int index = Integer.parseInt(indexAttr);
                     if (index < 0) {
                         error("'index' cannot be lower than 0", ele);
                     }else {
                         try {
                             this.parseState.push(new
ConstructorArgumentEntry(index));
```

```
//解析ele对应的属性元素
 Object value = parsePropertyValue(ele, bd, null);
 ConstructorArgumentValues.ValueHolder valueHolder =
new
ConstructorArgumentValues.ValueHolder(value);
 if (StringUtils.hasLength(typeAttr)) {
   valueHolder.setType(typeAttr);
}
 if (StringUtils.hasLength(nameAttr)) {
   valueHolder.setName(nameAttr);
}
valueHolder.setSource(extractSource(ele));
//不允许重复指定相同参数
 if (bd.getConstructorArgumentValues().
hasIndexedArgumentValue
 (index)) {
    error("Ambiguous constructor-arg entries for index " +
    index, ele);
 }else {
   (index, valueHolder);
   bd.getConstructorArgumentValues().
AddIndexedArgumentValue
}
}finally {
this.parseState.pop();
}
}
```

```
            }catch (NumberFormatException ex) {
                error("Attribute 'index' of tag 'constructor-arg' must be an
integer", ele);
            }
        }else {
            //没有index属性则忽略去属性，自动寻找
            try {
                this.parseState.push(new ConstructorArgumentEntry());
                Object value = parsePropertyValue(ele, bd, null);
                ConstructorArgumentValues.ValueHolder valueHolder = new
Constructor
               ArgumentValues.ValueHolder(value);
                if (StringUtils.hasLength(typeAttr)) {
                   valueHolder.setType(typeAttr);
               }
                if (StringUtils.hasLength(nameAttr)) {
                   valueHolder.setName(nameAttr);
               }
               valueHolder.setSource(extractSource(ele));

bd.getConstructorArgumentValues().addGenericArgumentValue(valueHolder
            }
            finally {
               this.parseState.pop();
            }
        }
    }
```

上面一段看似复杂的代码让很多人失去了耐心，但是，涉及的逻辑其实并不复杂，首先是提取constructor-arg上必要的属性（index、type、name）。

如果配置中指定了index属性，那么操作步骤如下。

（1）解析constructor-arg的子元素。

（2）使用ConstructorArgumentValues.ValueHolder类型来封装解析出来的元素。

（3）将type、name和index属性一并封装在ConstructorArgumentValues.ValueHolder类型中并添加至当前BeanDefinition的constructorArgumentValues的indexedArgumentValues属性中。

如果没有指定index属性，那么操作步骤如下。

（1）解析constructor-arg的子元素。

（2）使用ConstructorArgumentValues.ValueHolder类型来封装解析出来的元素。

（3）将type、name和index属性一并封装在ConstructorArgumentValues.ValueHolder类型中并添加至当前BeanDefinition的constructorArgumentValues的genericArgumentValues属性中。

可以看到，对于是否制定index属性来讲，Spring的处理流程是不同的，关键在于属性信息被保存的位置。

那么了解了整个流程后，我们尝试着进一步了解解析构造函数配置中子元素的过程，进入parsePropertyValue：

```
public Object parsePropertyValue(Element ele, BeanDefinition bd,
String propertyName) {
    String elementName = (propertyName != null) ?
        "<property> element for property '" + propertyName + "'" :
```

```
        "<constructor-arg> element";
    //一个属性只能对应一种类型：ref、value、list等
    NodeList nl = ele.getChildNodes();
    Element subElement = null;
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
       //对应description或者meta不处理
        if (node instanceof Element && !nodeNameEquals(node,
DESCRIPTION_ELEMENT) &&
            !nodeNameEquals(node, META_ELEMENT)) {
        if (subElement != null) {
            error(elementName + " must not contain more than one sub-
element", ele);
       }
        else {
            subElement = (Element) node;
       }
       }
    }
    //解析constructor-arg上的ref属性
    boolean hasRefAttribute = ele.hasAttribute(REF_ATTRIBUTE);
    //解析constructor-arg上的value属性
    boolean hasValueAttribute =
ele.hasAttribute(VALUE_ATTRIBUTE);
    if ((hasRefAttribute && hasValueAttribute) ||
        ((hasRefAttribute || hasValueAttribute) && subElement != null)) {
    /*
```

```
    * 在constructor-arg上不存在:
    * 1、同时既有ref属性又有value属性
    * 2、存在ref属性或者value属性且又有子元素
    */
   error(elementName +
     " is only allowed to contain either 'ref' attribute OR 'value'
     attribute OR sub-element", ele);
  }
   if (hasRefAttribute) {
    //ref属性的处理，使用RuntimeBeanReference封装对应的ref名
称
     String refName = ele.getAttribute(REF_ATTRIBUTE);
     if (!StringUtils.hasText(refName)) {
        error(elementName + " contains empty 'ref' attribute", ele);
    }
     RuntimeBeanReference ref = new
RuntimeBeanReference(refName);
    ref.setSource(extractSource(ele));
     return ref;
   }else if (hasValueAttribute) {
    //value属性的处理，使用TypedStringValue封装
     TypedStringValue valueHolder = new TypedStringValue
(ele.getAttribute
    (VALUE_ATTRIBUTE));
    valueHolder.setSource(extractSource(ele));
     return valueHolder;
   }else if (subElement != null) {
```

```
            //解析子元素
             return parsePropertySubElement(subElement, bd);
         }else {
            //既没有ref也没有value也没有子元素，Spring蒙圈了
             error(elementName + " must specify a ref or value", ele);
             return null;
         }
     }
```

从代码上来看，对构造函数中属性元素的解析，经历了以下几个过程。

（1）略过description或者meta。

（2）提取constructor-arg上的ref和value属性，以便于根据规则验证正确性，其规则为在constructor-arg上不存在以下情况。

同时既有ref属性又有value属性。

存在ref属性或者value属性且又有子元素。

（3）ref属性的处理。使用RuntimeBeanReference封装对应的ref名称，如：

```
<constructor-arg ref="a" >
```

（4）value属性的处理。使用TypedStringValue封装，例如：

```
<constructor-arg value="a" >
```

（5）子元素的处理。例如：

```
<constructor-arg>
  <map>
     <entry key="key" value="value" />
  </map>
</constructor-arg>
```

而对于子元素的处理，例如这里提到的在构造函数中又嵌入了子元

素 map 是怎么实现的呢？parsePropertySubElement中实现了对各种子元素的分类处理。

```
public Object parsePropertySubElement(Element ele, BeanDefinition bd) {
    return parsePropertySubElement(ele, bd, null);
}
public Object parsePropertySubElement(Element ele, BeanDefinition bd, String defaultValueType) {
    if (!isDefaultNamespace(ele)) {
        return parseNestedCustomElement(ele, bd);
    }
    else if (nodeNameEquals(ele, BEAN_ELEMENT)) {
        BeanDefinitionHolder nestedBd = parseBeanDefinitionElement(ele, bd);
        if (nestedBd != null) {
            nestedBd = decorateBeanDefinitionIfRequired(ele, nestedBd, bd);
        }
        return nestedBd;
    }
    else if (nodeNameEquals(ele, REF_ELEMENT)) {
        // A generic reference to any name of any bean.
        String refName = ele.getAttribute(BEAN_REF_ATTRIBUTE);
        boolean toParent = false;
        if (!StringUtils.hasLength(refName)) {
          //解析local
            refName = ele.getAttribute(LOCAL_REF_ATTRIBUTE);
```

```
                    if (!StringUtils.hasLength(refName)) {
                       //解析parent
                        refName = ele.getAttribute(PARENT_REF_ATTRIBUTE);
                        toParent = true;
                        if (!StringUtils.hasLength(refName)) {
                            error("'bean', 'local' or 'parent' is required for <ref>
element", ele);
                            return null;
                        }
                    }
                }
                if (!StringUtils.hasText(refName)) {
                    error("<ref> element contains empty target attribute", ele);
                    return null;
                }
                RuntimeBeanReference ref = new
RuntimeBeanReference(refName, toParent);
               ref.setSource(extractSource(ele));
                return ref;
            }
            //对idref元素的解析
            else if (nodeNameEquals(ele, IDREF_ELEMENT)) {
                return parseIdRefElement(ele);
            }
            //对value子元素的解析
            else if (nodeNameEquals(ele, VALUE_ELEMENT)) {
                return parseValueElement(ele, defaultValueType);
```

```
}
//对null子元素的解析
 else if (nodeNameEquals(ele, NULL_ELEMENT)) {
     // It's a distinguished null value. Let's wrap it in a
TypedStringValue
     // object in order to preserve the source location.
     TypedStringValue nullHolder = new TypedStringValue(null);
    nullHolder.setSource(extractSource(ele));
     return nullHolder;
}
 else if (nodeNameEquals(ele, ARRAY_ELEMENT)) {
    //解析array子元素
     return parseArrayElement(ele, bd);
}
 else if (nodeNameEquals(ele, LIST_ELEMENT)) {
    //解析list子元素
     return parseListElement(ele, bd);
}
 else if (nodeNameEquals(ele, SET_ELEMENT)) {
    //解析set子元素
     return parseSetElement(ele, bd);
}
 else if (nodeNameEquals(ele, MAP_ELEMENT)) {
    //解析map子元素
     return parseMapElement(ele, bd);
}
 else if (nodeNameEquals(ele, PROPS_ELEMENT)) {
```

```
            //解析props子元素
             return parsePropsElement(ele);
          }
          else {
             error("Unknown property sub-element: [" + ele.getNodeName() +
"]", ele);
             return null;
          }
       }
```

可以看到，在上面的函数中实现了所有可支持的子类的分类处理，到这里，我们已经大致理清构造函数的解析流程，至于再深入的解析读者有兴趣可以自己去探索。

7．解析子元素**property**

parsePropertyElement函数完成了对property属性的提取，property使用方式如下：

```
<bean id="test" class="test.TestClass">
    <property name="testStr" value="aaa"/>
</bean>
```

或者

```
<bean id="a">
    <property name="p">
       <list>
          <value>aa</value>
          <value>bb</value>
       </list>
    </property>
</bean>
```

而具体的解析过程如下：

```
public void parsePropertyElements(Element beanEle, BeanDefinition bd) {
    NodeList nl = beanEle.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        Node node = nl.item(i);
        if (isCandidateElement(node) && nodeNameEquals(node, PROPERTY_ELEMENT)) {
            parsePropertyElement((Element) node, bd);
        }
    }
}
```

有了之前分析构造函数的经验，这个函数我们并不难理解，无非是提取所有property的子元素，然后调用parsePropertyElement处理，parsePropertyElement代码如下：

```
public void parsePropertyElement(Element ele, BeanDefinition bd) {
    //获取配置元素中name的值
    String propertyName = ele.getAttribute(NAME_ATTRIBUTE);
    if (!StringUtils.hasLength(propertyName)) {
        error("Tag 'property' must have a 'name' attribute", ele);
        return;
    }
    this.parseState.push(new PropertyEntry(propertyName));
    try {
        //不允许多次对同一属性配置
        if (bd.getPropertyValues().contains(propertyName)) {
            error("Multiple 'property' definitions for property '" +
```

```
propertyName + "'", ele);
                return;
            }
             Object val = parsePropertyValue(ele, bd, propertyName);
             PropertyValue pv = new PropertyValue(propertyName, val);
             parseMetaElements(ele, pv);
            pv.setSource(extractSource(ele));
            bd.getPropertyValues().addPropertyValue(pv);
        }
        finally {
            this.parseState.pop();
        }
    }
```

可以看到上面函数与构造函数注入方式不同的是将返回值使用PropertyValue进行封装，并记录在了BeanDefinition中的propertyValues属性中。

8．解析子元素**qualifier**

对于qualifier元素的获取，我们接触更多的是注解的形式，在使用Spring框架中进行自动注入时，Spring容器中匹配的候选Bean数目必须有且仅有一个。当找不到一个匹配的Bean时， Spring容器将抛出BeanCreationException异常，并指出必须至少拥有一个匹配的Bean。

Spring允许我们通过Qualifier指定注入Bean的名称，这样歧义就消除了，而对于配置方式使用如：

```
<bean id="myTestBean" class="bean.MyTestBean">
    <qualifier
type="org.Springframework.beans.factory.annotation.Qualifier" value="qf"/>
</bean>
```

其解析过程与之前大同小异，这里不再重复叙述。

3.1.2 AbstractBeanDefinition属性

至此我们便完成了对XML文档到GenericBeanDefinition的转换，也就是说到这里，XML中所有的配置都可以在GenericBeanDefinition的实例类中找到对应的配置。

GenericBeanDefinition只是子类实现，而大部分的通用属性都保存在了AbstractBeanDefinition中，那么我们再次通过AbstractBeanDefinition的属性来回顾一下我们都解析了哪些对应的配置。

```
public abstract class AbstractBeanDefinition extends BeanMetadataAttributeAccessor
    implements BeanDefinition, Cloneable {
//此处省略静态变量以及final常量
  /**
    * bean的作用范围,对应bean属性scope
    */
   private String scope = SCOPE_DEFAULT;
  /**
    * 是否是单例,来自bean属性scope
    */
   private boolean singleton = true;
  /**
    * 是否是原型,来自bean属性scope
    */
   private boolean prototype = false;
  /**
    * 是否是抽象，对应bean属性abstract
```

```
   */
 private boolean abstractFlag = false;
/**
   * 是否延迟加载,对应bean属性lazy-init
   */
 private boolean lazyInit = false;
/**
* 自动注入模式,对应bean属性autowire
*/
private int autowireMode = AUTOWIRE_NO;
/**
  *依赖检查，Spring 3.0后弃用这个属性
  */
private int dependencyCheck = DEPENDENCY_CHECK_NONE;
/**
  * 用来表示一个bean的实例化依靠另一个bean先实例化,对应bean
属性depend-on
  */
private String[] dependsOn;
/**
  * autowire-candidate属性设置为false，这样容器在查找自动装配
对象时，
  * 将不考虑该bean，即它不会被考虑作为其他bean自动装配的候
选者，但是该bean本身还是可以使用自动
  装配来注入其他bean的。
  * 对应bean属性autowire-candidate
  */
```

```
private boolean autowireCandidate = true;
/**
 * 自动装配时当出现多个bean候选者时，将作为首选者,对应bean属性primary
 */
private boolean primary = false;
/**
 * 用于记录Qualifier，对应子元素qualifier
 */
private final Map<String, AutowireCandidateQualifier> qualifiers =
new LinkedHashMap<String, AutowireCandidateQualifier>(0);
/**
 * 允许访问非公开的构造器和方法，程序设置
 */
private boolean nonPublicAccessAllowed = true;
/**
 * 是否以一种宽松的模式解析构造函数，默认为true,
 * 如果为false,则在如下情况
 * interface ITest{}
 * class ITestImpl implements ITest{};
 * class Main{
 * Main(ITest i){}
 * Main(ITestImpl i){}
 * }
 * 抛出异常，因为Spring无法准确定位哪个构造函数
 * 程序设置
 */
```

```
private boolean lenientConstructorResolution = true;
/**
  * 记录构造函数注入属性，对应bean属性constructor-arg
  */
    private ConstructorArgumentValues constructorArgumentValues;
   /**
     * 普通属性集合
     */
    private MutablePropertyValues propertyValues;
   /**
     * 方法重写的持有者 ,记录lookup-method、replaced-method元
素
     */
    private MethodOverrides methodOverrides = new
MethodOverrides();
   /**
     * 对应bean属性factory-bean，用法：
     * <bean id="instanceFactoryBean"
class="example.chapter3.InstanceFactoryBean"/>
     * <bean id="currentTime" factory-bean="instanceFactoryBean"
factory-method="
     createTime"/>
     */
    private String factoryBeanName;
   /**
     * 对应bean属性factory-method
     */
```

private String factoryMethodName;
/**
* 初始化方法，对应bean属性init-method
*/
private String initMethodName;
/**
* 销毁方法，对应bean属性destory-method
*/
private String destroyMethodName;
/**
* 是否执行init-method，程序设置
*/
private boolean enforceInitMethod = true;
/**
* 是否执行destory-method，程序设置
*/
private boolean enforceDestroyMethod = true;
/**
* 是否是用户定义的而不是应用程序本身定义的,创建AOP时候为true，程序设置
*/
private boolean synthetic = false;
/**
* 定义这个bean的应用，APPLICATION：用户，INFRASTRUCTURE：完全内部使用，与用户无关，SUPPORT：
某些复杂配置的一部分
* 程序设置

```
        */
       private int role = BeanDefinition.ROLE_APPLICATION;
      /**
        * bean的描述信息
        */
       private String description;
      /**
        * 这个bean定义的资源
        */
       private Resource resource;
    //此处省略set/get方法
    }
```

3.1.3 解析默认标签中的自定义标签元素

到这里我们已经完成了分析默认标签的解析与提取过程，或许涉及的内容太多，我们已经忘了我们从哪个函数开始的了，我们再次回顾下默认标签解析函数的起始函数：

```
    protected void processBeanDefinition(Element ele,
BeanDefinitionParserDelegate delegate) {
        BeanDefinitionHolder bdHolder =
delegate.parseBeanDefinitionElement(ele);
        if (bdHolder != null) {
            bdHolder = delegate.decorateBeanDefinitionIfRequired(ele,
bdHolder);
            try {
                // Register the final decorated instance.
                BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
```

```
getReader
                  Context().getRegistry());
              }
              catch (BeanDefinitionStoreException ex) {
                  getReaderContext().error("Failed to register bean definition with
name '" +
                  bdHolder.getBeanName() + "'", ele, ex);
              }
              // Send registration event.
              getReaderContext().fireComponentRegistered(new
BeanComponentDefinition
       (bdHolder));
           }
       }
```

我们已经用了大量的篇幅分析了 BeanDefinitionHolder bdHolder = delegate.parseBean DefinitionElement(ele)这句代码，接下来，我们要进行 bdHolder = delegate.decorateBean DefinitionIfRequired(ele, bdHolder)代码的分析，首先大致了解下这句代码的作用，其实我们可以从语义上分析：如果需要的话就对beanDefinition进行装饰，那这句代码到底是什么功能呢？其实这句代码适用于这样的场景，如：

```
<bean id="test" class="test.MyClass">
    <mybean:user username="aaa"/>
</bean>
```

当Spring中的bean使用的是默认的标签配置，但是其中的子元素却使用了自定义的配置时，这句代码便会起作用了。可能有人会有疑问，之前讲过，对 bean 的解析分为两种类型，一种是默认类型的解析，另一种是自定义类型的解析，这不正是自定义类型的解析吗？为什么会在

默认类型解析中单独添加一个方法处理呢？确实，这个问题很让人迷惑，但是，不知道聪明的读者是否有发现，这个自定义类型并不是以Bean 的形式出现的呢？我们之前讲过的两种类型的不同处理只是针对Bean 的，这里我们看到，这个自定义类型其实是属性。好了，我们继续分析下这段代码的逻辑。

```
public BeanDefinitionHolder
decorateBeanDefinitionIfRequired(Element ele, BeanDefinitionHolder
definitionHolder) {
    return decorateBeanDefinitionIfRequired(ele, definitionHolder, null);
}
```

这里将函数中第三个参数设置为空，那么第三个参数是做什么用的呢？什么情况下不为空呢？其实这第三个参数是父类 bean， 当对某个嵌套配置进行分析时，这里需要传递父类beanDefinition。分析源码得知这里传递的参数其实是为了使用父类的scope属性，以备子类若没有设置scope时默认使用父类的属性，这里分析的是顶层配置，所以传递null。将第三个参数设置为空后进一步跟踪函数：

```
public BeanDefinitionHolder decorateBeanDefinitionIfRequired(
    Element ele, BeanDefinitionHolder definitionHolder, BeanDefinition
containingBd) {
        BeanDefinitionHolder finalDefinition = definitionHolder;
        NamedNodeMap attributes = ele.getAttributes();
      //遍历所有的属性，看看是否有适用于修饰的属性
        for (int i = 0; i < attributes.getLength(); i++) {
            Node node = attributes.item(i);
            finalDefinition =decorateIfRequired(node, finalDefinition,
containingBd);
        }
```

```
        NodeList children = ele.getChildNodes();
       //遍历所有的子节点，看看是否有适用于修饰的子元素
          for (int i = 0; i < children.getLength(); i++) {
             Node node = children.item(i);
             if (node.getNodeType() == Node.ELEMENT_NODE) {
                finalDefinition =decorateIfRequired(node, finalDefinition,
containingBd);
             }
          }
          return finalDefinition;
       }
```

上面的代码，我们看到函数分别对元素的所有属性以及子节点进行了 decorateIfRequired函数的调用，我们继续跟踪代码：

```
    private BeanDefinitionHolder decorateIfRequired(
        Node node, BeanDefinitionHolder originalDef, BeanDefinition
containingBd) {
    //获取自定义标签的命名空间
        String namespaceUri = getNamespaceURI(node);
    //对于非默认标签进行修饰
        if (!isDefaultNamespace(namespaceUri)) {
    //根据命名空间找到对应的处理器
           NamespaceHandler handler = this.readerContext.
getNamespaceHandler Resolver().
          resolve(namespaceUri);
           if (handler != null) {
       //进行修饰
           return handler.decorate(node, originalDef, new
```

```
ParserContext(this.readerContext,
                this, containingBd));
            }
            else if (namespaceUri != null && namespaceUri.startsWith("http:
//www.
                Springframework.org/")) {
                error("Unable to locate Spring NamespaceHandler for XML
schema namespace
                [" + namespaceUri + "]", node);
            }
            else {
                // A custom namespace, not to be handled by Spring - maybe
"xml:...".
                if (logger.isDebugEnabled()) {
                    logger.debug("No Spring NamespaceHandler found for XML
schema
                    namespace [" + namespaceUri + "]");
                }
            }
        }
        return originalDef;
    }
    public String getNamespaceURI(Node node) {
        return node.getNamespaceURI();
    }
    public boolean isDefaultNamespace(String namespaceUri) {
        //BEANS_NAMESPACE_URI =
```

"http://www.Springframework.org/schema/beans";
            return (!StringUtils.hasLength(namespaceUri) ||
BEANS_NAMESPACE_URI.equals
           (namespaceUri));
        }

程序走到这里，条理其实已经非常清楚了，首先获取属性或者元素的命名空间，以此来判断该元素或者属性是否适用于自定义标签的解析条件，找出自定义类型所对应的NamespaceHandler并进行进一步解析。在自定义标签解析的章节我们会重点讲解，这里暂时先略过。

我们总结下decorateBeanDefinitionIfRequired方法的作用，在decorateBeanDefinitionIfRequired中我们可以看到对于程序默认的标签的处理其实是直接略过的，因为默认的标签到这里已经被处理完了，这里只对自定义的标签或者说对 bean 的自定义属性感兴趣。在方法中实现了寻找自定义标签并根据自定义标签寻找命名空间处理器，并进行进一步的解析。

3.1.4 注册解析的BeanDefinition

对于配置文件，解析也解析完了，装饰也装饰完了，对于得到的beanDinition已经可以满足后续的使用要求了，唯一还剩下的工作就是注册了，也就是processBeanDefinition函数中的BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder, getReaderContext().getRegistry())代码的解析了。

public static void registerBeanDefinition(
BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
throws BeanDefinitionStoreException {
    //使用beanName做唯一标识注册

```
        String beanName = definitionHolder.getBeanName();
        registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());
    //注册所有的别名
        String[] aliases = definitionHolder.getAliases();
        if (aliases != null) {
            for (String aliase : aliases) {
                registry.registerAlias(beanName, aliase);
            }
        }
    }
```

从上面的代码可以看出，解析的beanDefinition都会被注册到BeanDefinitionRegistry类型的实例registry中，而对于beanDefinition的注册分成了两部分：通过beanName的注册以及通过别名的注册。

1．通过beanName注册BeanDefinition

对于 beanDefinition 的注册，或许很多人认为的方式就是将 beanDefinition 直接放入 map中就好了，使用 beanName 作为 key。确实，Spring 就是这么做的，只不过除此之外，它还做了点别的事情。

```
    public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
    throws BeanDefinitionStoreException {
        Assert.hasText(beanName, "Bean name must not be empty");
        Assert.notNull(beanDefinition, "BeanDefinition must not be null");
        if (beanDefinition instanceof AbstractBeanDefinition) {
            try {
                /*
                 * 注册前的最后一次校验，这里的校验不同于之前的XML
```

```
文件校验，
                * 主要是对于AbstractBeanDefinition属性中的
methodOverrides校验，
                * 校验methodOverrides是否与工厂方法并存或者
methodOverrides对应的方法根本不存在
                */
               ((AbstractBeanDefinition) beanDefinition).validate();
            }
            catch (BeanDefinitionValidationException ex) {
               throw new BeanDefinitionStoreException (beanDefinition.
getResource
               Description(), beanName,
               "Validation of bean definition failed", ex);
            }
         }
         //因为beanDefinitionMap是全局变量，这里定会存在并发访问的
情况
         synchronized (this.beanDefinitionMap) {
            Object oldBeanDefinition =
this.beanDefinitionMap.get(beanName);
            //处理注册已经注册的beanName情况
            if (oldBeanDefinition != null) {
              //如果对应的BeanName已经注册且在配置中配置了bean不允
许被覆盖，则抛出异常。
               if (!this.allowBeanDefinitionOverriding) {
                  throw new BeanDefinitionStoreException(beanDefinition.
getResource
```

```
                    Description(), beanName,
                    "Cannot register bean definition [" + beanDefinition +
                    "] for bean '" + beanName +
                    "': There is already [" + oldBeanDefinition + "] bound.");
                }else {
                    if (this.logger.isInfoEnabled()) {
                        this.logger.info("Overriding bean definition for bean '" +
beanName +
                        "': replacing [" + oldBeanDefinition + "] with ["
                        + beanDefinition + "]");
                    }
                }
            }else {
                //记录beanName
                this.beanDefinitionNames.add(beanName);
                 this.frozenBeanDefinitionNames = null;
            }
            //注册beanDefinition
             this.beanDefinitionMap.put(beanName, beanDefinition);
        }
        //重置所有beanName对应的缓存
        resetBeanDefinition(beanName);
    }
```

上面的代码中我们看到，在对于bean的注册处理方式上，主要进行了几个步骤。

（1）对AbstractBeanDefinition的校验。在解析XML文件的时候我们提过校验，但是此校验非彼校验，之前的校验时针对于 XML 格式的

校验，而此时的校验时针是对于 AbstractBean Definition的methodOverrides属性的。

（2）对beanName已经注册的情况的处理。如果设置了不允许bean的覆盖，则需要抛出异常，否则直接覆盖。

（3）加入map缓存。

（4）清除解析之前留下的对应beanName的缓存。

2．通过别名注册BeanDefinition

在理解了注册bean的原理后，理解注册别名的原理就容易多了。

```
public void registerAlias(String name, String alias) {
    Assert.hasText(name, "'name' must not be empty");
    Assert.hasText(alias, "'alias' must not be empty");
   //如果beanName与alias相同的话不记录alias,并删除对应的alias
    if (alias.equals(name)) {
      this.aliasMap.remove(alias);
    }else {
      //如果alias不允许被覆盖则抛出异常
       if (!allowAliasOverriding()) {
          String registeredName = this.aliasMap.get(alias);
          if (registeredName != null && !registeredName.equals(name)) {
             throw new IllegalStateException("Cannot register alias '" +
alias
             + "' for name '" +
             name + "': It is already registered for name '" +
             registeredName + "'.");
          }
      }
      //当A->B存在时，若再次出现A->C->B时候则会抛出异常
```

```
            checkForAliasCircle(name, alias);
            this.aliasMap.put(alias, name);
        }
    }
```

由以上代码中可以得知注册alias的步骤如下。

（1）alias与beanName相同情况处理。若alias与beanName并名称相同则不需要处理并删除掉原有alias。

（2）alias覆盖处理。若aliasName已经使用并已经指向了另一beanName则需要用户的设置进行处理。

（3）alias循环检查。当A->B存在时，若再次出现A->C->B时候则会抛出异常。

（4）注册alias。

3.1.5 通知监听器解析及注册完成

通过代码getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder))完成此工作，这里的实现只为扩展，当程序开发人员需要对注册BeanDefinition事件进行监听时可以通过注册监听器的方式并将处理逻辑写入监听器中，目前在 Spring 中并没有对此事件做任何逻辑处理。

3.2 alias标签的解析

通过上面较长的篇幅我们终于分析完了默认标签中对 bean 标签的处理，那么我们之前提到过，对配置文件的解析包括对import标签、alias标签、bean标签、beans标签的处理，现在我们已经完成了最重要也是最核心的功能，其他的解析步骤也都是围绕第3个解析而进行的。在分析了第3个解析步骤后，再回过头来看看对alias标签的解析。

在对bean进行定义时，除了使用id属性来指定名称之外，为了提供

多个名称，可以使用alias标签来指定。而所有的这些名称都指向同一个bean，在某些情况下提供别名非常有用，比如为了让应用的每一个组件能更容易地对公共组件进行引用。

然而，在定义 bean 时就指定所有的别名并不是总是恰当的。有时我们期望能在当前位置为那些在别处定义的 bean 引入别名。在 XML 配置文件中，可用单独的<alias/>元素来完成bean别名的定义。如配置文件中定义了一个JavaBean：

```
<bean id="testBean" class="com.test"/>
```

要给这个JavaBean增加别名，以方便不同对象来调用。我们就可以直接使用bean标签中的name属性：

```
<bean id="testBean" name="testBean,testBean2" class="com.test"/>
```

同样，Spring还有另外一种声明别名的方式：

```
<bean id="testBean" class="com.test"/>
<alias name="testBean" alias="testBean,testBean2"/>
```

考虑一个更为具体的例子，组件 A 在 XML 配置文件中定义了一个名为 componentA 的DataSource类型的bean，但组件B却想在其XML文件中以componentB命名来引用此bean。而且在主程序MyApp的XML配置文件中，希望以myApp的名字来引用此bean。最后容器加载3个XML文件来生成最终的ApplicationContext。在此情形下，可通过在配置文件中添加下列alias元素来实现：

```
<alias name="componentA" alias="componentB"/>
<alias name="componentA" alias="myApp" />
```

这样一来，每个组件及主程序就可通过唯一名字来引用同一个数据源而互不干扰。

在之前的章节已经讲过了对于bean中name元素的解析，那么我们现在再来深入分析下对于alias标签的解析过程。

```
protected void processAliasRegistration(Element ele) {
```

```
//获取beanName
 String name = ele.getAttribute(NAME_ATTRIBUTE);
//获取alias
 String alias = ele.getAttribute(ALIAS_ATTRIBUTE);
 boolean valid = true;
 if (!StringUtils.hasText(name)) {
     getReaderContext().error("Name must not be empty", ele);
     valid = false;
 }
 if (!StringUtils.hasText(alias)) {
     getReaderContext().error("Alias must not be empty", ele);
     valid = false;
 }
 if (valid) {
     try {
        //注册alias
         getReaderContext().getRegistry().registerAlias(name, alias);
     }
     catch (Exception ex) {
         getReaderContext().error("Failed to register alias '" + alias +
         "' for bean with name '" + name + "'", ele, ex);
     }
     //别名注册后通知监听器做相应处理
     getReaderContext().fireAliasRegistered(name, alias,
extractSource(ele));
 }
}
```

可以发现，跟之前讲过的bean中的alias解析大同小异，都是将别名与beanName组成一对注册至registry中。这里不再赘述。

3.3 import标签的解析

对于Spring配置文件的编写，我想，经历过庞大项目的人，都有那种恐惧的心理，太多的配置文件了。不过，分模块是大多数人能想到的方法，但是，怎么分模块，那就仁者见仁，智者见智了。使用import是个好办法，例如我们可以构造这样的Spring配置文件：

```
applicationContext.xml
<?xml version="1.0" encoding="gb2312"?>
<!DOCTYPE beans PUBLIC "-//Spring//DTD BEAN//EN"
"http://www.Springframework.org/
dtd/Spring-beans.dtd">
<beans>
    <import resource="customerContext.xml" />
    <import resource="systemContext.xml" />
    ... ...
</beans>
```

applicationContext.xml文件中使用import的方式导入有模块配置文件，以后若有新模块的加入，那就可以简单修改这个文件了。这样大大简化了配置后期维护的复杂度，并使配置模块化，易于管理。我们来看看Spring是如何解析import配置文件的呢？

```
protected void importBeanDefinitionResource(Element ele) {
    //获取resource属性
    String location = ele.getAttribute(RESOURCE_ATTRIBUTE);
    //如果不存在resource属性则不做任何处理
```

```
            if (!StringUtils.hasText(location)) {
                getReaderContext().error("Resource location must not be empty",
ele);
               return;
            }
            //解析系统属性，格式如： "${user.dir}"
            location = environment.resolveRequiredPlaceholders(location);
            Set<Resource> actualResources = new LinkedHashSet<Resource>
(4);
           //判定location是决定URI还是相对URI
            boolean absoluteLocation = false;
            try {
                absoluteLocation = ResourcePatternUtils.isUrl(location) ||
ResourceUtils.toURI
               (location).isAbsolute();
            }
            catch (URISyntaxException ex) {
                // cannot convert to an URI, considering the location relative
                // unless it is the well-known Spring prefix "classpath*:"
            }
            // Absolute or relative?
           //如果是绝对URI则直接根据地址加载对应的配置文件
            if (absoluteLocation) {
                try {
                    int importCount =
getReaderContext().getReader().loadBeanDefinitions
                    (location, actualResources);
```

```
                if (logger.isDebugEnabled()) {
                    logger.debug("Imported " + importCount + " bean definitions
from
                    URL location [" + location + "]");
                }
            }
            catch (BeanDefinitionStoreException ex) {
                getReaderContext().error(
                "Failed to import bean definitions from URL location [" +
                location + "]", ele, ex);
            }
        }
        else {
            //如果是相对地址则根据相对地址计算出绝对地址
            try {
                int importCount;
                //Resource存在多个子实现类，如VfsResource、
FileSystemResource等，
                //而每个resource的createRelative方式实现都不一样，所以这
里先使用子类的方法
                尝试解析
                Resource relativeResource = getReaderContext().
getResource(). Create
                Relative(location);
                if (relativeResource.exists()) {
                    importCount = getReaderContext().getReader().
loadBeanDefinitions
```

```
(relativeResource);
actualResources.add(relativeResource);
}else {
//如果解析不成功，则使用默认的解析器
ResourcePatternResolver进行解析
String baseLocation =
getReaderContext().getResource().getURL().
toString();
importCount =
getReaderContext().getReader().loadBeanDefinitions(
StringUtils.applyRelativePath(baseLocation, location),
actualResources);
}
if (logger.isDebugEnabled()) {
logger.debug("Imported " + importCount + " bean definitions
from
relative location [" + location + "]");
}
}
catch (IOException ex) {
getReaderContext().error("Failed to resolve current resource
location", ele, ex);
}
catch (BeanDefinitionStoreException ex) {
getReaderContext().error("Failed to import bean definitions
from
relative location [" + location + "]",
```

```
                ele, ex);
            }
        }
        //解析后进行监听器激活处理
         Resource[] actResArray = actualResources.toArray(new
Resource[actualResources. size()]);
         getReaderContext().fireImportProcessed(location, actResArray,
extractSource(ele));
    }
```

上面的代码不难，相信配合注释会很好理解，我们总结一下大致流程便于读者更好地梳理，在解析<import标签时，Spring进行解析的步骤大致如下。

（1）获取resource属性所表示的路径。

（2）解析路径中的系统属性，格式如“${user.dir}”。

（3）判定location是绝对路径还是相对路径。

（4）如果是绝对路径则递归调用bean的解析过程，进行另一次的解析。

（5）如果是相对路径则计算出绝对路径并进行解析。

（6）通知监听器，解析完成。

3.4 嵌入式beans标签的解析

对于嵌入式的beans标签，相信大家使用过或者至少接触过，非常类似于import标签所提供的功能，使用如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www.Springframework.
            org/schema/beans/Spring-beans.xsd">
             <bean id="aa" class="test.aa"/>
            <beans>
            </beans>
        </beans>
```

对于嵌入式 beans 标签来讲，并没有太多可讲，与单独的配置文件并没有太大的差别，无非是递归调用 beans 的解析过程，相信读者根据之前讲解过的内容已经有能力理解其中的奥秘了。

第**4**章　自定义标签的解析

在之前的章节中，我们提到了在Spring中存在默认标签与自定义标签两种，而在上一章节中我们分析了Spring中对默认标签的解析过程，相信大家一定已经有所感悟。那么，现在将开始新的里程，分析Spring中自定义标签的加载过程。同样，我们还是先再次回顾一下，当完成从配置文件到Document的转换并提取对应的root后，将开始了所有元素的解析，而在这一过程中便开始了默认标签与自定义标签两种格式的区分，函数如下：

```
        protected void parseBeanDefinitions(Element root,
BeanDefinitionParserDelegate delegate) {
             if (delegate.isDefaultNamespace(root)) {
                 NodeList nl = root.getChildNodes();
                 for (int i = 0; i < nl.getLength(); i++) {
                     Node node = nl.item(i);
```

```
                if (node instanceof Element) {
                    Element ele = (Element) node;
                    if (delegate.isDefaultNamespace(ele)) {
                        parseDefaultElement(ele, delegate);
                    }
                    else {
                        delegate.parseCustomElement(ele);
                    }
                }
            }
        }
        else {
            delegate.parseCustomElement(root);
        }
    }
```

在本章中，所有的功能都是围绕其中的一句代码delegate.parseCustomElement(root)开展的。从上面的函数我们可以看出，当Spring拿到一个元素时首先要做的是根据命名空间进行解析，如果是默认的命名空间，则使用 parseDefaultElement 方法进行元素解析，否则使用 parseCustom Element 方法进行解析。在分析自定义标签的解析过程前，我们先了解一下自定义标签的使用过程。

4.1 自定义标签使用

在很多情况下，我们需要为系统提供可配置化支持，简单的做法可以直接基于Spring的标准 bean 来配置，但配置较为复杂或者需要更多丰富控制的时候，会显得非常笨拙。一般的做法会用原生态的方式去解析

定义好的XML文件，然后转化为配置对象。这种方式当然可以解决所有问题，但实现起来比较繁琐，特别是在配置非常复杂的时候，解析工作是一个不得不考虑的负担。Spring提供了可扩展Schema的支持，这是一个不错的折中方案，扩展Spring自定义标签配置大致需要以下几个步骤（前提是要把Spring的Core包加入项目中）。

创建一个需要扩展的组件。

定义一个XSD文件描述组件内容。

创建一个文件，实现BeanDefinitionParser接口，用来解析XSD文件中的定义和组件定义。

创建一个Handler文件，扩展自NamespaceHandlerSupport，目的是将组件注册到Spring容器。

编写Spring.handlers和Spring.schemas文件。

现在我们就按照上面的步骤带领读者一步步地体验自定义标签的过程。

（1）首先我们创建一个普通的POJO，这个POJO 没有任何特别之处，只是用来接收配置文件。

```
package test.customtag;
public class User {
    private String userName;
    private String email;
   //省略set/get方法
}
```

（2）定义一个XSD文件描述组件内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.lexueba.com/schema/user"
xmlns:tns="http://www.lexueba.com/schema/user"
```

```
elementFormDefault="qualified">
<element name="user">
   <complexType>
       <attribute name="id" type="string"/>
       <attribute name="userName" type="string"/>
       <attribute name="email" type="string"/>
   </complexType>
</element>
</schema>
```

在上面的 XSD 文件中描述了一个新的 targetNamespace，并在这个空间中定义了一个 name为user的element，user有3个属性id、userName和email，其中email的类型为string。这3个类主要用于验证 Spring配置文件中自定义格式。XSD文件是XML DTD的替代者，使用XML Schema语言进行编写，这里对XSD Schema不做太多解释，有兴趣的读者可以参考相关的资料。

（3）创建一个文件，实现BeanDefinitionParser接口，用来解析XSD文件中的定义和组件定义。

```
package test.customtag;
Import
org.Springframework.beans.factory.support.BeanDefinitionBuilder;
import
org.Springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.Springframework.util.StringUtils;
import org.w3c.dom.Element;
public class UserBeanDefinitionParser extends
AbstractSingleBeanDefinitionParser {
   //Element对应的类
```

```
    protected Class getBeanClass(Element element) {
        return User.class;
    }
    //从element中解析并提取对应的元素
    protected void doParse(Element element, BeanDefinitionBuilder
bean) {
        String userName = element.getAttribute("userName");
        String email = element.getAttribute("email");
          //将提取的数据放入到BeanDefinitionBuilder中，待到完成所
有bean的解析后统一注册到
          beanFactory中
        if (StringUtils.hasText(userName)) {
            bean.addPropertyValue("userName", userName);
        }
        if (StringUtils.hasText(email)) {
            bean.addPropertyValue("email", email);
        }
    }
}
```

（4）创建一个 Handler 文件，扩展自 NamespaceHandlerSupport，目的是将组件注册到Spring容器。

```
package test.customtag;
import
org.Springframework.beans.factory.xml.NamespaceHandlerSupport;
public class MyNamespaceHandler extends NamespaceHandlerSupport
{
    public void init() {
```

```
        registerBeanDefinitionParser("user", new
UserBeanDefinitionParser());
    }
}
```

以上代码很简单，无非是当遇到自定义标签<user:aaa这样类似于以user开头的元素，就会把这个元素扔给对应的UserBeanDefinitionParser去解析。

（5）编写Spring.handlers和Spring.schemas文件，默认位置是在工程的/META-INF/文件夹下，当然，你可以通过Spring的扩展或者修改源码的方式改变路径。

Spring.handlers。

```
http\://www.lexueba.com/schema/user=test.customtag.MyNamespaceHa
```

Spring.schemas。

```
http\://www.lexueba.com/schema/user.xsd=META-INF/Spring-test.xsd
```

到这里，自定义的配置就结束了，而Spring加载自定义的大致流程是遇到自定义标签然后就去Spring.handlers和Spring.schemas中去找对应的handler和XSD，默认位置是/META-INF/下，进而有找到对应的handler以及解析元素的Parser，从而完成了整个自定义元素的解析，也就是说自定义与 Spring 中默认的标准配置不同在于 Spring 将自定义标签解析的工作委托给了用户去实现。

（6）创建测试配置文件，在配置文件中引入对应的命名空间以及 XSD 后，便可以直接使用自定义标签了。

```
<beans xmlns="http://www.Springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:myname="http://www.lexueba.com/schema/user"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
```

```
http://www.
    Springframework.org/schema/beans/Spring-beans-2.0.xsd
  http://www.lexueba.com/schema/user
http://www.lexueba.com/schema/user.xsd">
  <myname:user id="testbean" userName="aaa" email="bbb"/>
  </beans>
```

（7）测试。

```
public static void main(String[] args) {
    ApplicationContext bf = new ClassPathXmlApplicationContext
("test/customtag/
    test.xml");
    User user=(User) bf.getBean("testbean");
    System.out.println(user.getUserName()+","+user.getEmail());
}
```

不出意外的话，你应该看到了我们期待的结果，控制台上打印出了：

aaa,bbb

在上面的例子中，我们实现了通过自定义标签实现了通过属性的方式将 user 类型的 Bean赋值，在Spring中自定义标签非常常用，例如我们熟知的事务标签：tx(<tx:annotation-driven>)。

4.2 自定义标签解析

了解了自定义标签的使用后，我们带着强烈的好奇心来探究一下自定义标签的解析过程。

```
public BeanDefinition parseCustomElement(Element ele) {
    return parseCustomElement(ele, null);
```

```
    }
    //containingBd为父类bean，对顶层元素的解析应设置为null
    public BeanDefinition parseCustomElement(Element ele, BeanDefinition containingBd) {
        //获取对应的命名空间
        String namespaceUri = getNamespaceURI(ele);
        //根据命名空间找到对应的NamespaceHandler
        NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().
        resolve(namespaceUri);
        if (handler == null) {
            error("Unable to locate Spring NamespaceHandler for XML schema namespace ["
            + namespaceUri + "]", ele);
            return null;
        }
        //调用自定义的NamespaceHandler进行解析
        return handler.parse(ele, new ParserContext(this.readerContext, this,
        containingBd));
    }
```

相信了解了自定义标签的使用方法后，或多或少会对自定义标签的实现过程有一个自己的想法。其实思路非常的简单，无非是根据对应的bean 获取对应的命名空间，根据命名空间解析对应的处理器，然后根据用户自定义的处理器进行解析。可是有些事情说起来简单做起来难，我们先看看如何获取命名空间吧。

4.2.1 获取标签的命名空间

标签的解析是从命名空间的提起开始的，无论是区分 Spring 中默认标签和自定义标签还是区分自定义标签中不同标签的处理器都是以标签所提供的命名空间为基础的，而至于如何提取对应元素的命名空间其实并不需要我们亲自去实现，在 org.w3c.dom.Node 中已经提供了方法供我们直接调用：

```
public String getNamespaceURI(Node node) {
    return node.getNamespaceURI();
}
```

4.2.2 提取自定义标签处理器

有了命名空间，就可以进行NamespaceHandler的提取了，继续之前的parseCustomElement函数的跟踪，分析 NamespaceHandler handler = this.readerContext.getNamespaceHandlerResolver().
resolve(namespaceUri)，在readerContext初始化的时候其属性namespaceHandlerResolver已经被初始化为了DefaultNamespaceHandlerResolver 的实例，所以，这里调用的 resolve 方法其实调用的是DefaultNamespaceHandlerResolver 类中的方法。我们进入 DefaultNamespaceHandlerResolver 的resolve方法进行查看。

DefaultNamespaceHandlerResolver.java

```
public NamespaceHandler resolve(String namespaceUri) {
   //获取所有已经配置的handler映射
    Map<String, Object> handlerMappings = getHandlerMappings();
   //根据命名空间找到对应的信息
    Object handlerOrClassName = handlerMappings.get(namespaceUri);
    if (handlerOrClassName == null) {
        return null;
    }else if (handlerOrClassName instanceof NamespaceHandler) {
```

```
//已经做过解析的情况，直接从缓存读取
 return (NamespaceHandler) handlerOrClassName;
}else {
//没有做过解析，则返回的是类路径
 String className = (String) handlerOrClassName;
 try {
  //使用反射将类路径转化为类
   Class<?> handlerClass = ClassUtils.forName(className,
this.classLoader);
   if (!NamespaceHandler.class.isAssignableFrom(handlerClass))
{
     throw new FatalBeanException("Class [" + className + "]
for
     namespace [" + namespaceUri +
     "] does not implement the [" + NamespaceHandler. class.
     getName() + "] interface");
  }
  //初始化类
   NamespaceHandler namespaceHandler = (NamespaceHandler)
BeanUtils.
  instantiateClass(handlerClass);
  //调用自定义的NamespaceHandler的初始化方法
  namespaceHandler.init();
  //记录在缓存
   handlerMappings.put(namespaceUri, namespaceHandler);
   return namespaceHandler;
 }catch (ClassNotFoundException ex) {
```

```
                throw new FatalBeanException("NamespaceHandler class [" +
className +
                "] for namespace [" +
                namespaceUri + "] not found", ex);
            }catch (LinkageError err) {
                throw new FatalBeanException("Invalid NamespaceHandler
class [" +
                className + "] for namespace [" +
                namespaceUri + "]: problem with handler class file or dependent
                class", err);
            }
        }
    }
```

上面的函数清晰地阐述了解析自定义 NamespaceHandler 的过程，通过之前的示例程序我们了解到如果要使用自定义标签，那么其中一项必不可少的操作就是在Spring.handlers文件中配置命名空间与命名空间处理器的映射关系。只有这样，Spring才能根据映射关系找到匹配的处理器，而寻找匹配的处理器就是在上面函数中实现的，当获取到自定义的 NamespaceHandler 之后就可以进行处理器初始化并解析了。我们不妨再次回忆一下示例中对于命名空间处理器的内容：

```
    public class MyNamespaceHandler extends NamespaceHandlerSupport
{
        public void init() {
            registerBeanDefinitionParser("user", new
UserBeanDefinitionParser());
        }
    }
```

当得到自定义命名空间处理后会马上执行 namespaceHandler.init()来进行自定义 Bean DefinitionParser的注册。在这里，你可以注册多个标签解析器，当前示例中只有支持<myname:user的写法，你也可以在这里注册多个解析器，如<myname:A、<myname:B等，使得myname的命名空间中可以支持多种标签解析。

注册后，命名空间处理器就可以根据标签的不同来调用不同的解析器进行解析。那么，根据上面的函数与之前介绍过的例子，我们基本上可以推断getHandlerMappings的主要功能就是读取Spring.handlers配置文件并将配置文件缓存在map中。

```
private Map<String, Object> getHandlerMappings() {
  //如果没有被缓存则开始进行缓存
   if (this.handlerMappings == null) {
      synchronized (this) {
         if (this.handlerMappings == null) {
            try {
               //this.handlerMappingsLocation 在构造函数中已经被初始化为:META-
               INF/Spring.handlers
                Properties mappings =
                PropertiesLoaderUtils.loadAllProperties (this.
                handlerMappingsLocation, this.classLoader);
                if (logger.isDebugEnabled()) {
                   logger.debug("Loaded NamespaceHandler mappings: " + mappings);
               }
                Map<String, Object> handlerMappings = new ConcurrentHashMap<
```

```
String, Object>();
            //将Properties格式文件合并到Map格式的handlerMappings中
            CollectionUtils.mergePropertiesIntoMap(mappings, handlerMappings);
            this.handlerMappings = handlerMappings;
          }
          catch (IOException ex) {
            throw new IllegalStateException(
            "Unable to load NamespaceHandler mappings from
            location [" + this.handlerMappingsLocation + "]", ex);
          }
        }
      }
    }
    return this.handlerMappings;
}
```

同我们想象的一样，借助了工具类PropertiesLoaderUtils对属性handlerMappingsLocation进行了配置文件的读取，handlerMappingsLocation被默认初始化为“META-INF/Spring.handlers”。

4.2.3 标签解析

得到了解析器以及要分析的元素后，Spring就可以将解析工作委托给自定义解析器去解析了。在Spring中的代码为：

```
return handler.parse(ele, new ParserContext(this.readerContext, this, containingBd))
```

以之前提到的示例进行分析，此时的 handler 已经被实例化成为了

我们自定义的MyNamespaceHandler了，而MyNamespaceHandler也已经完成了初始化的工作，但是在我们实现的自定义命名空间处理器中并没有实现 parse 方法，所以推断，这个方法是父类中的实现，查看父类NamespaceHandlerSupport中的parse方法。

NamespaceHandlerSupport.java

public BeanDefinition parse(Element element, ParserContext parserContext) {

//寻找解析器并进行解析操作

returnfindParserForElement(element, parserContext).parse(element, parserContext);

}

解析过程中首先是寻找元素对应的解析器，进而调用解析器中的 parse 方法，那么结合示例来讲，其实就是首先获取在MyNameSpaceHandler类中的init方法中注册的对应的UserBean DefinitionParser实例，并调用其parse方法进行进一步解析。

private BeanDefinitionParser findParserForElement(Element element, ParserContext parser

Context) {

//获取元素名称，也就是<myname:user中的user,若在示例中，此时localName为user

String localName = parserContext.getDelegate().getLocalName(element);

//根据user找到对应的解析器，也就是在

//registerBeanDefinitionParser("user", new UserBeanDefinitionParser());

//注册的解析器

BeanDefinitionParser parser = this.parsers.get(localName);

```
    if (parser == null) {
      parserContext.getReaderContext().fatal(
       "Cannot locate BeanDefinitionParser for element [" + localName +
       "]", element);
    }
    return parser;
}
```

而对于parse方法的处理：

```
public final BeanDefinition parse(Element element, ParserContext
parserContext) {
    AbstractBeanDefinition definition = parseInternal(element,
parserContext);
    if (definition != null && !parserContext.isNested()) {
       try {
          String id = resolveId(element, definition, parserContext);
          if (!StringUtils.hasText(id)) {
             parserContext.getReaderContext().error(
              "Id is required for element '" + parserContext.
             getDelegate().getLocalName(element)
              + "' when used as a top-level tag", element);
          }
          String[] aliases = new String[0];
          String name = element.getAttribute(NAME_ATTRIBUTE);
          if (StringUtils.hasLength(name)) {
             aliases =

StringUtils.trimArrayElements(StringUtils.commaDelimitedListToStringArra
```

```
            }
            //将AbstractBeanDefinition转换为BeanDefinitionHolder并注册
             BeanDefinitionHolder holder = new
BeanDefinitionHolder(definition, id,
            aliases);
             registerBeanDefinition(holder, parserContext.getRegistry());
             if (shouldFireEvents()) {
               //需要通知监听器则进行处理
                BeanComponentDefinition componentDefinition = new
BeanComponent
               Definition(holder);
               postProcessComponentDefinition(componentDefinition);
               parserContext.registerComponent(componentDefinition);
            }
        }
         catch (BeanDefinitionStoreException ex) {
             parserContext.getReaderContext().error(ex.getMessage(),
element);
             return null;
        }
    }
     return definition;
}
```

虽说是对自定义配置文件的解析，但是，我们可以看到，在这个函数中大部分的代码是用来处理将解析后的AbstractBeanDefinition转化为BeanDefinitionHolder并注册的功能，而真正去做解析的事情委托给了函数parseInternal，正是这句代码调用了我们自定义的解析函数。

在parseInternal中并不是直接调用自定义的doParse函数，而是进行了一系列的数据准备，包括对beanClass、scope、lazyInit等属性的准备。

```
protected final AbstractBeanDefinition parseInternal(Element element,
ParserContext
    parserContext) {
        BeanDefinitionBuilder builder =
BeanDefinitionBuilder.genericBeanDefinition();
        String parentName = getParentName(element);
        if (parentName != null) {
          builder.getRawBeanDefinition().setParentName(parentName);
        }
    //获取自定义标签中的class，此时会调用自定义解析器如
UserBeanDefinitionParser中的getBeanClass
    方法。
        Class<?> beanClass = getBeanClass(element);
        if (beanClass != null) {
          builder.getRawBeanDefinition().setBeanClass(beanClass);
        }
        else {
       //若子类没有重写getBeanClass方法则尝试检查子类是否重写
getBeanClassName方法
            String beanClassName = getBeanClassName(element);
            if (beanClassName != null) {

builder.getRawBeanDefinition().setBeanClassName(beanClassName);
            }
```

```
            }
builder.getRawBeanDefinition().setSource(parserContext.extractSource(elem
             if (parserContext.isNested()) {
            //若存在父类则使用父类的scope属性
builder.setScope(parserContext.getContainingBeanDefinition().getScope());
            }
             if (parserContext.isDefaultLazyInit()) {
                 // Default-lazy-init applies to custom bean definitions as well.
            //配置延迟加载
                builder.setLazyInit(true);
            }
        //调用子类重写的doParse方法进行解析
            doParse(element, parserContext, builder);
             return builder.getBeanDefinition();
            }
        protected void doParse(Element element, ParserContext parserContext,
BeanDefinition
        Builder builder) {
            doParse(element, builder);
        }
```

回顾一下全部的自定义标签处理过程，虽然在实例中我们定义UserBeanDefinitionParser，但是在其中我们只是做了与自己业务逻辑相关的部分。不过我们没做但是并不代表没有，在这个处理过程中同样也是按照Spring中默认标签的处理方式进行，包括创建BeanDefinition以及进行相应默认属性的设置，对于这些工作Spring都默默地帮我们实现

了，只是暴露出一些接口来供用户实现个性化的业务。通过对本章的了解，相信读者对Spring中自定义标签的使用以及在解析自定义标签过程中Spring为我们做了哪些工作会有一个全面的了解。到此为止我们已经完成了Spring中全部的解析工作，也就是说到现在为止我们已经理解了Spring将bean从配置文件到加载到内存中的全过程，而接下来的任务便是如何使用这些 bean，下一章将介绍 bean的加载。

第5章　bean的加载

经过前面的分析，我们终于结束了对XML配置文件的解析，接下来将会面临更大的挑战，就是对bean加载的探索。bean加载的功能实现远比bean的解析要复杂得多，同样，我们还是以本书开篇的示例为基础，对于加载bean的功能，在Spring中的调用方式为：

```
MyTestBean bean=(MyTestBean) bf.getBean("myTestBean")
```

这句代码实现了什么样的功能呢？我们可以先快速体验一下Spring中代码是如何实现的。

```
public Object getBean(String name) throws BeansException {
    return doGetBean(name, null, null, false);
}
protected <T> T doGetBean(
final String name, final Class<T> requiredType, final Object[] args, boolean
typeCheckOnly) throws BeansException {
   //提取对应的beanName
    final String beanName = transformedBeanName(name);
    Object bean;
   /*
```

* 检查缓存中或者实例工厂中是否有对应的实例

* 为什么首先会使用这段代码呢,

* 因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，

* Spring创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提早曝光

* 也就是将ObjectFactory加入到缓存中，一旦下个bean创建时候需要依赖上个bean则直接使用

ObjectFactory

*/

//直接尝试从缓存获取或者singletonFactories中的ObjectFactory中获取

```
Object sharedInstance = getSingleton(beanName);
if (sharedInstance != null && args == null) {
    if (logger.isDebugEnabled()) {
        if (isSingletonCurrentlyInCreation(beanName)) {
            logger.debug("Returning eagerly cached instance of singleton
bean
            '" + beanName +
            "' that is not fully initialized yet - a consequence of
            a circular reference");
        }
        else {
            beanName + "'");
            logger.debug("Returning cached instance of singleton bean '"
+
        }
```

}
//返回对应的实例，有时候存在诸如BeanFactory的情况并不是直接返回实例本身而是返回指定方法返回的实例
bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
}else {
//只有在单例情况才会尝试解决循环依赖，原型模式情况下，如果存在
//A中有B的属性，B中有A的属性，那么当依赖注入的时候，就会产生当A还未创建完的时候因为
//对于B的创建再次返回创建A，造成循环依赖，也就是下面的情况
//isPrototypeCurrentlyInCreation(beanName)为true
if (isPrototypeCurrentlyInCreation(beanName)) {
throw new BeanCurrentlyInCreationException(beanName);
}
BeanFactory parentBeanFactory = getParentBeanFactory();
//如果 beanDefinitionMap 中也就是在所有已经加载的类中不包括 beanName 则尝试从
parentBeanFactory中检测
if (parentBeanFactory != null && !containsBeanDefinition(beanName)) {
String nameToLookup = originalBeanName(name);
//递归到BeanFactory中寻找
if (args != null) {
return (T) parentBeanFactory.getBean(nameToLookup, args);
}

```
            else {
                return parentBeanFactory.getBean(nameToLookup,
requiredType);
            }
        }
        //如果不是仅仅做类型检查则是创建bean，这里要进行记录
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }
        //将存储XML配置文件的GernericBeanDefinition转换为
RootBeanDefinition，如果指
        定BeanName是子Bean的话同时会合并父类的相关属性
        final RootBeanDefinition mbd =
getMergedLocalBeanDefinition(beanName);
        checkMergedBeanDefinition(mbd, beanName, args);
        String[] dependsOn = mbd.getDependsOn();
        //若存在依赖则需要递归实例化依赖的bean
        if (dependsOn != null) {
            for (String dependsOnBean : dependsOn) {
                getBean(dependsOnBean);
                //缓存依赖调用
                registerDependentBean(dependsOnBean, beanName);
            }
        }
        //实例化依赖的bean后便可以实例化mbd本身了
        //singleton模式的创建
        if (mbd.isSingleton()) {
```

```
                    sharedInstance = getSingleton(beanName, new
ObjectFactory<Object>() {
                        public Object getObject() throws BeansException {
                        try {
                            return createBean(beanName, mbd, args);
                        }
                        catch (BeansException ex) {
                           destroySingleton(beanName);
                            throw ex;
                        }
                    }
                });
                bean = getObjectForBeanInstance(sharedInstance, name,
beanName, mbd);
            }else if (mbd.isPrototype()) {
                //prototype模式的创建(new)
                Object prototypeInstance = null;
                try {
                   beforePrototypeCreation(beanName);
                    prototypeInstance = createBean(beanName, mbd, args);
                }
                finally {
                   afterPrototypeCreation(beanName);
                }
                bean = getObjectForBeanInstance(prototypeInstance, name,
beanName, mbd);
            }else {
```

```
            //指定的scope上实例化bean
             String scopeName = mbd.getScope();
             final Scope scope = this.scopes.get(scopeName);
             if (scope == null) {
                throw new IllegalStateException("No Scope registered for scope
'"
                + scopeName + "'");
            }
             try {
                Object scopedInstance = scope.get(beanName, new
ObjectFactory<Object>() {
                    public Object getObject() throws BeansException {
                       beforePrototypeCreation(beanName);
                        try {
                            return createBean(beanName, mbd, args);
                       }
                        finally {
                           afterPrototypeCreation(beanName);
                       }
                   }
               });
                bean = getObjectForBeanInstance(scopedInstance, name,
beanName, mbd);
            }
             catch (IllegalStateException ex) {
                throw new BeanCreationException(beanName,
                thread; " +
```

```
                "Scope '" + scopeName + "' is not active for the current
                "consider defining a scoped proxy for this bean if you
                intend to refer to it from a singleton",
               ex);
            }
        }
    }
     //检查需要的类型是否符合bean的实际类型
     if (requiredType != null && bean != null &&
!requiredType.isAssignableFrom
     (bean.getClass())) {
        try {
            return getTypeConverter().convertIfNecessary(bean,
requiredType);
        }
        catch (TypeMismatchException ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Failed to convert bean '" + name + "' to
required type
                [" +
                ClassUtils.getQualifiedName(requiredType) + "]", ex);
            }
            throw new BeanNotOfRequiredTypeException(name,
requiredType, bean.
           getClass());
        }
    }
```

```
    return (T) bean;
}
```

仅从代码量上就能看出来 bean 的加载经历了一个相当复杂的过程，其中涉及各种各样的考虑。相信读者细心阅读上面的代码，并参照部分代码注释，是可以粗略地了解整个Spring加载bean的过程。对于加载过程中所涉及的步骤大致如下。

（1）转换对应beanName。

或许很多人不理解转换对应 beanName 是什么意思，传入的参数name 不就是 beanName吗？其实不是，这里传入的参数可能是别名，也可能是 FactoryBean，所以需要进行一系列的解析，这些解析内容包括如下内容。

去除FactoryBean的修饰符，也就是如果name="&aa"，那么会首先去除&而使name="aa"。

取指定alias所表示的最终beanName，例如别名A指向名称为B的bean则返回B；若别名A指向别名B，别名B又指向名称为C的bean则返回C。

（2）尝试从缓存中加载单例。

单例在Spring的同一个容器内只会被创建一次，后续再获取bean，就直接从单例缓存中获取了。当然这里也只是尝试加载，首先尝试从缓存中加载，如果加载不成功则再次尝试从singletonFactories中加载。因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖，在Spring中创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提早曝光加入到缓存中，一旦下一个bean创建时候需要依赖上一个bean则直接使用ObjectFactory（后面章节会对循环依赖重点讲解）。

（3）bean的实例化。

如果从缓存中得到了bean的原始状态，则需要对bean进行实例化。

这里有必要强调一下，缓存中记录的只是最原始的bean 状态，并不一定是我们最终想要的bean。举个例子，假如我们需要对工厂bean进行处理，那么这里得到的其实是工厂bean的初始状态，但是我们真正需要的是工厂bean中定义的factory-method方法中返回的bean，而getObjectForBeanInstance就是完成这个工作的，后续会详细讲解。

（4）原型模式的依赖检查。

只有在单例情况下才会尝试解决循环依赖，如果存在A中有B的属性，B中有A的属性，那么当依赖注入的时候，就会产生当A还未创建完的时候因为对于B的创建再次返回创建A，造成循环依赖，也就是情况：isPrototypeCurrentlyInCreation(beanName)判断true。

（5）检测parentBeanFactory。

从代码上看，如果缓存没有数据的话直接转到父类工厂上去加载了，这是为什么呢？

可能读者忽略了一个很重要的判断条件：parentBeanFactory != null && !containsBean Definition (beanName)，parentBeanFactory != null。parentBeanFactory如果为空，则其他一切都是浮云，这个没什么说的，但是!containsBeanDefinition(beanName)就比较重要了，它是在检测如果当前加载的XML配置文件中不包含beanName所对应的配置，就只能到parentBeanFactory去尝试下了，然后再去递归的调用getBean方法。

（6）将存储XML配置文件的GernericBeanDefinition转换为RootBeanDefinition。

因为从XML配置文件中读取到的Bean信息是存储在GernericBeanDefinition中的，但是所有的Bean 后续处理都是针对于RootBeanDefinition 的，所以这里需要进行一个转换，转换的同时如果父类bean不为空的话，则会一并合并父类的属性。

（7）寻找依赖。

因为 bean 的初始化过程中很可能会用到某些属性，而某些属性很

可能是动态配置的，并且配置成依赖于其他的bean，那么这个时候就有必要先加载依赖的bean，所以，在Spring 的加载顺寻中，在初始化某一个bean的时候首先会初始化这个bean所对应的依赖。

（8）针对不同的scope进行bean的创建。

我们都知道，在Spring中存在着不同的scope，其中默认的是singleton，但是还有些其他的配置诸如prototype、request之类的。在这个步骤中，Spring会根据不同的配置进行不同的初始化策略。

（9）类型转换。

程序到这里返回bean后已经基本结束了，通常对该方法的调用参数requiredType是为空的，但是可能会存在这样的情况，返回的 bean 其实是个 String，但是 requiredType 却传入 Integer类型，那么这时候本步骤就会起作用了，它的功能是将返回的bean转换为requiredType所指定的类型。当然，String转换为Integer是最简单的一种转换，在Spring中提供了各种各样的转换器，用户也可以自己扩展转换器来满足需求。

经过上面的步骤后bean的加载就结束了，这个时候就可以返回我们所需要的bean了，图5-1直观地反映了整个过程。其中最重要的就是步骤（8），针对不同的scope进行bean的创建，你会看到各种常用的Spring特性在这里的实现。

在细化分析各个步骤提供的功能前，我们有必要先了解下FactoryBean的用法。

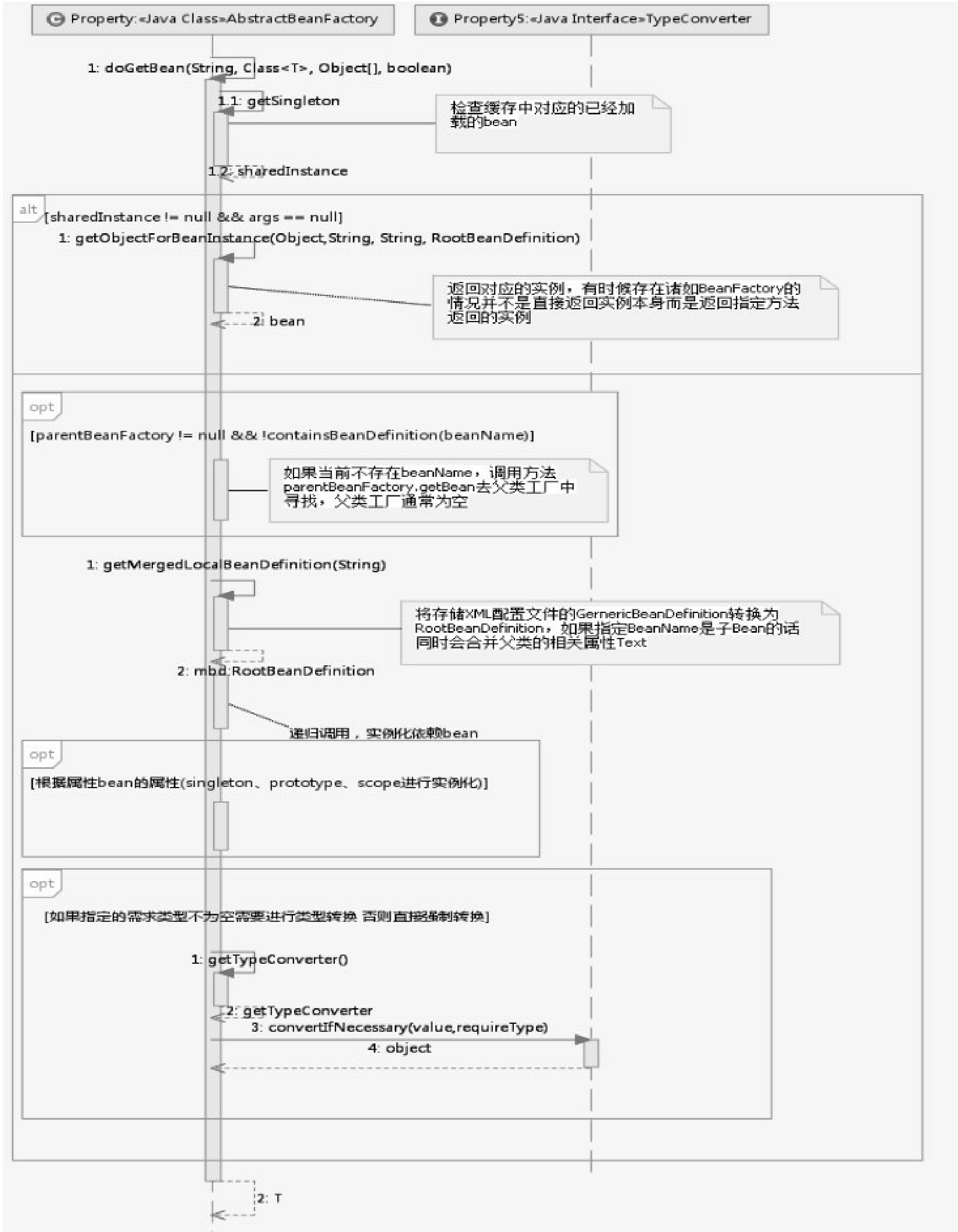


图5-1 bean的获取过程

5.1 FactoryBean的使用

一般情况下，Spring通过反射机制利用bean的class属性指定实现类来实例化bean 。在某些情况下，实例化 bean 过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息，配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring为此提供了一个org.Springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化bean的逻辑。

FactoryBean 接口对于 Spring 框架来说占有重要的地位，Spring 自身就提供了 70 多个FactoryBean的实现。它们隐藏了实例化一些复杂bean的细节，给上层应用带来了便利。从Spring 3.0开始， FactoryBean开始支持泛型，即接口声明改为FactoryBean<T>的形式：

```
package org.Springframework.beans.factory;
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<?> getObjectType();
    boolean isSingleton();
}
```

在该接口中还定义了以下3个方法。

T getObject()：返回由FactoryBean创建的 bean实例，如果isSingleton()返回 true，则该实例会放到Spring容器中单实例缓存池中。

boolean isSingleton()：返回由FactoryBean创建的 bean实例的作用域是 singleton还是prototype。

Class<T> getObjectType()：返回FactoryBean创建的bean类型。

当配置文件中<bean>的 class属性配置的实现类是 FactoryBean时，通过 getBean()方法返回的不是 FactoryBean 本身，而是FactoryBean#getObject()方法所返回的对象，相当于

FactoryBean#getObject()代理了getBean()方法。例如：如果使用传统方式配置下面Car 的<bean>时，Car的每个属性分别对应一个<property>元素标签。

```
public class Car {
    private int maxSpeed ;
    private String brand ;
    private double price ;
//get/set方法
}
```

如果用FactoryBean 的方式实现就会灵活一些，下例通过逗号分割符的方式一次性地为Car的所有属性指定配置值：

```
public class CarFactoryBean implements FactoryBean<Car> {
    private String carInfo ;
    public Car getObject () throws Exception {
        Car car = new Car () ;
        String [] infos = carInfo .split ( "," ) ;
        car.setBrand ( infos [ 0 ]) ;
        car.setMaxSpeed ( Integer. valueOf ( infos [ 1 ])) ;
        car.setPrice ( Double. valueOf ( infos [ 2 ])) ;
        return car;
    }
    public Class<Car> getObjectType () {
        return Car. class ;
    }
    public boolean isSingleton () {
        return false ;
    }
```

```
    public String getCarInfo () {
        return this . carInfo ;
    }
    // 接受逗号分割符设置属性信息
    public void setCarInfo ( String carInfo ) {
        this . carInfo = carInfo;
    }
}
```

有了这个 CarFactoryBean 后，就可以在配置文件中使用下面这种自定义的配置方式配置Car Bean了：

```
<bean id="car" class="com.test.factorybean.CarFactoryBean" carInfo="超级跑车,400,2000000"/>
```

当调用 getBean("car") 时，Spring 通过反射机制发现 CarFactoryBean 实现了 FactoryBean的接口，这时Spring容器就调用接口方法CarFactoryBean#getObject()方法返回。如果希望获取CarFactoryBean的实例，则需要在使用getBean(beanName) 方法时在beanName前显示的加上"&" 前缀，例如getBean("&car")。

5.2 缓存中获取单例bean

介绍过FactoryBean的用法后，我们就可以了解bean加载的过程了。前面已经提到过，单例在Spring的同一个容器内只会被创建一次，后续再获取bean直接从单例缓存中获取，当然这里也只是尝试加载，首先尝试从缓存中加载，然后再次尝试尝试从singletonFactories中加载。因为在创建单例bean的时候会存在依赖注入的情况，而在创建依赖的时候为了避免循环依赖， Spring创建bean的原则是不等bean创建完成就会将创建bean的ObjectFactory提早曝光加入到缓存中，一旦下一个bean创建时

需要依赖上个bean，则直接使用ObjectFactory。

```
public Object getSingleton(String beanName) {
    //参数true设置标识允许早期依赖
return getSingleton(beanName, true);
}
protected Object getSingleton(String beanName, boolean
allowEarlyReference) {
    //检查缓存中是否存在实例
    Object singletonObject = this.singletonObjects.get(beanName);
    if (singletonObject == null) {
        //如果为空，则锁定全局变量并进行处理
        synchronized (this.singletonObjects) {
            //如果此bean正在加载则不处理
            singletonObject = this.earlySingletonObjects.get(beanName);
            if (singletonObject == null && allowEarlyReference) {
                //当某些方法需要提前初始化的时候则会调用
addSingletonFactory 方法将对应
                的ObjectFactory初始化策略存储在singletonFactories
                (beanName);
                ObjectFactory singletonFactory = this.singletonFactories.get
                if (singletonFactory != null) {
                    //调用预先设定的getObject方法
                    singletonObject = singletonFactory.getObject();
                    //记录在缓存中，earlySingletonObjects和
singletonFactories互斥
                    this.earlySingletonObjects.put(beanName,
singletonObject);
```

```
                    this.singletonFactories.remove(beanName);
                }
            }
        }
    }
    return (singletonObject != NULL_OBJECT ? singletonObject : null);
}
```

这个方法因为涉及循环依赖的检测，以及涉及很多变量的记录存取，所以让很多读者摸不着头脑。这个方法首先尝试从singletonObjects里面获取实例，如果获取不到再从earlySingleton Objects 里面获取，如果还获取不到，再尝试从 singletonFactories 里面获取 beanName 对应的ObjectFactory，然后调用这个 ObjectFactory 的 getObject 来创建 bean，并放到 earlySingleton Objects里面去，并且从singletonFacotories里面remove掉这个ObjectFactory，而对于后续的所有内存操作都只为了循环依赖检测时候使用，也就是在allowEarlyReference为true的情况下才会使用。

这里涉及用于存储bean的不同的map，可能让读者感到崩溃，简单解释如下。

singletonObjects：用于保存BeanName和创建bean实例之间的关系，bean name --> bean instance。

singletonFactories：用于保存BeanName和创建 bean的工厂之间的关系，bean name -->ObjectFactory。

earlySingletonObjects：也是保存 BeanName 和创建 bean 实例之间的关系，与singletonObjects的不同之处在于，当一个单例bean被放到这里面后，那么当bean还在创建过程中，就可以通过getBean方法获取到了，其目的是用来检测循环引用。

registeredSingletons：用来保存当前所有已注册的bean。

5.3 从bean的实例中获取对象

在 getBean 方法中，getObjectForBeanInstance 是个高频率使用的方法，无论是从缓存中获得bean还是根据不同的scope策略加载bean。总之，我们得到bean的实例后要做的第一步就是调用这个方法来检测一下正确性，其实就是用于检测当前 bean 是否是 FactoryBean类型的bean，如果是，那么需要调用该bean对应的FactoryBean实例中的getObject()作为返回值。

无论是从缓存中获取到的bean还是通过不同的scope策略加载的bean都只是最原始的bean状态，并不一定是我们最终想要的bean。举个例子，假如我们需要对工厂bean进行处理，那么这里得到的其实是工厂bean 的初始状态，但是我们真正需要的是工厂 bean 中定义的factory-method方法中返回的bean，而getObjectForBeanInstance方法就是完成这个工作的。

```
protected Object getObjectForBeanInstance(
Object beanInstance, String name, String beanName,
RootBeanDefinition mbd) {
    //如果指定的name是工厂相关(以&为前缀)且beanInstance又不是
FactoryBean类型则验证不通过
    if (BeanFactoryUtils.isFactoryDereference(name) && !(beanInstance
instanceof
    FactoryBean)) {
        throw new
BeanIsNotAFactoryException(transformedBeanName(name), beanInstance.
        getClass());
    }
```

```
        //现在我们有了个bean的实例，这个实例可能会是正常的bean或
者是FactoryBean
        //如果是 FactoryBean 我们使用它创建实例，但是如果用户想要直
接获取工厂实例而不是工厂的
        getObject方法对应的实例那么传入的name应该加入前缀&
         if (!(beanInstance instanceof FactoryBean) || BeanFactoryUtils.
IsFactory
         Dereference(name)) {
             return beanInstance;
        }
        //加载FactoryBean
         Object object = null;
         if (mbd == null) {
            //尝试从缓存中加载bean
             object = getCachedObjectForFactoryBean(beanName);
        }
         if (object == null) {
            //到这里已经明确知道beanInstance一定是FactoryBean类型
             FactoryBean<?> factory = (FactoryBean<?>) beanInstance;
            //containsBeanDefinition检测beanDefinitionMap中也就是在所有
已经加载的类中检测
            是否定义beanName
             if (mbd == null && containsBeanDefinition(beanName)) {
                //将存储XML配置文件的GernericBeanDefinition转换为
RootBeanDefinition，
                如果指定BeanName是子Bean的话同时会合并父类的相关属
性
```

```
            mbd = getMergedLocalBeanDefinition(beanName);
        }
        //是否是用户定义的而不是应用程序本身定义的
        boolean synthetic = (mbd != null && mbd.isSynthetic());
        object = getObjectFromFactoryBean(factory, beanName,
!synthetic);
    }
    return object;
}
```

从上面的代码来看，其实这个方法并没有什么重要的信息，大多是些辅助代码以及一些功能性的判断，而真正的核心代码却委托给了getObjectFromFactoryBean，我们来看看getObjectForBeanInstance中的所做的工作。

（1）对FactoryBean正确性的验证。

（2）对非FactoryBean不做任何处理。

（3）对bean进行转换。

（4）将从Factory中解析bean的工作委托给getObjectFromFactoryBean。

```
protected Object getObjectFromFactoryBean(FactoryBean factory,
String beanName, boolean
shouldPostProcess) {
    //如果是单例模式
    if (factory.isSingleton() && containsSingleton(beanName)) {
        synchronized (getSingletonMutex()) {
            Object object = this.factoryBeanObjectCache.get(beanName);
            if (object == null) {
                shouldPostProcess);
```

```
                    object = doGetObjectFromFactoryBean(factory, beanName,
                    this.factoryBeanObjectCache.put(beanName, (object != null ?
object :
                   NULL_OBJECT));
                }
                return (object != NULL_OBJECT ? object : null);
            }
        }
        else {
            return doGetObjectFromFactoryBean(factory, beanName,
shouldPostProcess);
        }
    }
```

很遗憾，在这个代码中我们还是没有看到想要看到的代码，在这个方法里只做了一件事情，就是返回的 bean 如果是单例的，那就必须要保证全局唯一，同时，也因为是单例的，所以不必重复创建，可以使用缓存来提高性能，也就是说已经加载过就要记录下来以便于下次复用，否则的话就直接获取了。

在doGetObjectFromFactoryBean方法中我们终于看到了我们想要看到的方法，也就是object =factory.getObject()，是的，就是这句代码，我们的历程犹如剥洋葱一样，一层一层的直到最内部的代码实现，虽然很简单。

```
    private Object doGetObjectFromFactoryBean(
    final FactoryBean factory, final String beanName, final boolean
shouldPostProcess)
    throws BeanCreationException {
        Object object;
```

```
try {
    //需要权限验证
    if (System.getSecurityManager() != null) {
        AccessControlContext acc = getAccessControlContext();
        try {
            Object>() {
            object = AccessController.doPrivileged(new
PrivilegedExceptionAction<
                public Object run() throws Exception {
                    return factory.getObject();
                }
            }, acc);
        }
        catch (PrivilegedActionException pae) {
            throw pae.getException();
        }
    }
    else {
        //直接调用getObject方法
        object = factory.getObject();
    }
}
catch (FactoryBeanNotInitializedException ex) {
    throw new BeanCurrentlyInCreationException(beanName,
ex.toString());
}
catch (Throwable ex) {
```

```
            throw new BeanCreationException(beanName, "FactoryBean
threw exception on
            object creation", ex);
        }
        if (object == null && isSingletonCurrentlyInCreation(beanName)) {
            throw new BeanCurrentlyInCreationException(
            beanName, "FactoryBean which is currently in creation returned
null
            from getObject");
        }
        if (object != null && shouldPostProcess) {
            try {
              //调用ObjectFactory的后处理器
               object = postProcessObjectFromFactoryBean(object,
beanName);
           }
            catch (Throwable ex) {
               throw new BeanCreationException(beanName, "Post-processing
of the
    FactoryBean's object failed", ex);
           }
        }
        return object;
    }
```

上面我们已经讲述了FactoryBean的调用方法，如果bean声明为FactoryBean类型，则当提取bean时提取的并不是FactoryBean，而是FactoryBean中对应的getObject方法返回的bean，而

doGetObjectFromFactoryBean正是实现这个功能的。但是，我们看到在上面的方法中除了调用 object = factory.getObject()得到我们想要的结果后并没有直接返回，而是接下来又做了些后处理的操作，这个又是做什么用的呢？于是我们跟踪进入AbstractAutowireCapableBeanFactory类的postProcessObjectFromFactoryBean方法：

AbstractAutowireCapableBeanFactory.java

```
protected Object postProcessObjectFromFactoryBean(Object object, String beanName) {
    return applyBeanPostProcessorsAfterInitialization(object, beanName);
}
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
throws BeansException {
    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        result = beanProcessor.postProcessAfterInitialization(result, beanName);
        if (result == null) {
            return result;
        }
    }
    return result;
}
```

对于后处理器的使用我们还未过多接触，后续章节会使用大量篇幅介绍，这里，我们只需了解在Spring获取bean的规则中有这样一条：尽可能保证所有bean初始化后都会调用注册的BeanPostProcessor 的

postProcessAfterInitialization 方法进行处理，在实际开发过程中大可以针对此特性设计自己的业务逻辑。

5.4 获取单例

之前我们讲解了从缓存中获取单例的过程，那么，如果缓存中不存在已经加载的单例bean就需要从头开始bean的加载过程了，而Spring中使用getSingleton的重载方法实现bean的加载过程。

```
public Object getSingleton(String beanName, ObjectFactory
singletonFactory) {
    Assert.notNull(beanName, "'beanName' must not be null");
    //全局变量需要同步
    synchronized (this.singletonObjects) {
      //首先检查对应的bean是否已经加载过，因为singleton模式其实
就是复用以创建的bean，所
      以这一步是必须的
      Object singletonObject = this.singletonObjects.get(beanName);
      //如果为空才可以进行singleto的bean的初始化
      if (singletonObject == null) {
        if (this.singletonsCurrentlyInDestruction) {
          throw new BeanCreationNotAllowedException(beanName,
          "Singleton bean creation not allowed while the singletons
          of this factory are in destruction " +
          "(Do not request a bean from a BeanFactory in a destroy
          method implementation!)");
        }
        if (logger.isDebugEnabled()) {
```

```
                logger.debug("Creating shared instance of singleton bean '" +
beanName + "'");
            }
            beforeSingletonCreation(beanName);
            boolean recordSuppressedExceptions =
(this.suppressedExceptions == null);
            if (recordSuppressedExceptions) {
                this.suppressedExceptions = new LinkedHashSet<Exception>
();
            }
            try {
               //初始化bean
                singletonObject = singletonFactory.getObject();
            }
            catch (BeanCreationException ex) {
                if (recordSuppressedExceptions) {
                    for (Exception suppressedException :
this.suppressedExceptions) {
                        ex.addRelatedCause(suppressedException);
                    }
                }
                throw ex;
            }
            finally {
                if (recordSuppressedExceptions) {
                    this.suppressedExceptions = null;
                }
```

```
                afterSingletonCreation(beanName);
            }
            //加入缓存
            addSingleton(beanName, singletonObject);
        }
        return (singletonObject != NULL_OBJECT ? singletonObject : null);
    }
}
```

上述代码中其实是使用了回调方法，使得程序可以在单例创建的前后做一些准备及处理操作，而真正的获取单例bean的方法其实并不是在此方法中实现的，其实现逻辑是在ObjectFactory类型的实例singletonFactory中实现的。而这些准备及处理操作包括如下内容。

（1）检查缓存是否已经加载过。

（2）若没有加载，则记录beanName的正在加载状态。

（3）加载单例前记录加载状态。

可能你会觉得beforeSingletonCreation方法是个空实现，里面没有任何逻辑，但其实不是，这个函数中做了一个很重要的操作：记录加载状态，也就是通过 this.singletonsCurrentlyIn Creation.add(beanName)将当前正要创建的bean记录在缓存中，这样便可以对循环依赖进行检测。

```
protected void beforeSingletonCreation(String beanName) {
    if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletons
    CurrentlyInCreation.add(beanName)) {
        throw new BeanCurrentlyInCreationException(beanName);
    }
}
```

（4）通过调用参数传入的ObjectFactory的个体Object方法实例化bean。

（5）加载单例后的处理方法调用。

同步骤（3）的记录加载状态相似，当bean加载结束后需要移除缓存中对该bean的正在加载状态的记录。

```
protected void afterSingletonCreation(String beanName) {
    if (!this.inCreationCheckExclusions.contains(beanName) &&
!this.singletons
    CurrentlyInCreation.remove(beanName)) {
        throw new IllegalStateException("Singleton '" + beanName + "'
isn't currently
        in creation");
    }
}
```

（6）将结果记录至缓存并删除加载bean过程中所记录的各种辅助状态。

```
protected void addSingleton(String beanName, Object singletonObject)
{
    synchronized (this.singletonObjects) {
        this.singletonObjects.put(beanName, (singletonObject != null ?
singletonObject :
        NULL_OBJECT));
        this.singletonFactories.remove(beanName);
        this.earlySingletonObjects.remove(beanName);
        this.registeredSingletons.add(beanName);
    }
}
```

（7）返回处理结果。

虽然我们已经从外部了解了加载bean的逻辑架构，但现在我们还并没有开始对bean加载功能的探索，之前提到过， bean 的加载逻辑其实是在传入的 ObjectFactory 类型的参数singletonFactory中定义的，我们反推参数的获取，得到如下代码：

```
sharedInstance = getSingleton(beanName, new ObjectFactory<Object>
() {
    public Object getObject() throws BeansException {
        try {
            return createBean(beanName, mbd, args);
        }
        catch (BeansException ex) {
            destroySingleton(beanName);
            throw ex;
        }
    }
});
```

ObjectFactory的核心部分其实只是调用了createBean的方法，所以我们还需要到createBean方法中追寻真理。

5.5 准备创建bean

我们不可能指望在一个函数中完成一个复杂的逻辑，而且我们跟踪了这么多Spring代码，经历了这么多函数，或多或少也发现了一些规律：一个真正干活的函数其实是以 do 开头的，比如doGetObjectFromFactoryBean；而给我们错觉的函数，比如getObjectFromFactoryBean，其实只是从全局角度去做些统筹的工作。这

个规则对于createBean也不例外，那么让我们看看在createBean函数中做了哪些准备工作。

```
protected Object createBean(final String beanName, final
RootBeanDefinition mbd, final
    Object[] args) throws BeanCreationException {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating instance of bean '" + beanName + "'");
        }
        //锁定class,根据设置的class属性或者根据className来解析Class
        resolveBeanClass(mbd, beanName);
        //验证及准备覆盖的方法
        try {
            mbd.prepareMethodOverrides();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new
BeanDefinitionStoreException(mbd.getResourceDescription(),
            beanName, "Validation of method overrides failed", ex);
        }
        try {
            //给BeanPostProcessors一个机会来返回代理来替代真正的实例
            Object bean = resolveBeforeInstantiation(beanName, mbd);
            if (bean != null) {
                return bean;
            }
        }
        catch (Throwable ex) {
```

```
            throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
            "BeanPostProcessor before instantiation of bean failed", ex);
        }
        Object beanInstance = doCreateBean(beanName, mbd, args);
        if (logger.isDebugEnabled()) {
            logger.debug("Finished creating instance of bean '" + beanName +
"'");
        }
        return beanInstance;
    }
```

从代码中我们可以总结出函数完成的具体步骤及功能。

（1）根据设置的class属性或者根据className来解析Class。

（2）对override属性进行标记及验证。

很多读者可能会不知道这个方法的作用，因为在 Spring 的配置里面根本就没有诸如override-method之类的配置，那么这个方法到底是干什么用的呢？

其实在 Spring 中确实没有 override-method 这样的配置，但是如果读过前面的部分，可能会有所发现，在Spring配置中是存在lookup-method和replace-method的，而这两个配置的加载其实就是将配置统一存放在BeanDefinition中的methodOverrides属性里，而这个函数的操作其实也就是针对于这两个配置的。

（3）应用初始化前的后处理器，解析指定bean是否存在初始化前的短路操作。

（4）创建bean。

我们首先查看下对override属性标记及验证的逻辑实现。

5.5.1 处理ovverride属性

查看源码中AbstractBeanDefinition类的prepareMethodOverrides方法：

```
public void prepareMethodOverrides() throws
BeanDefinitionValidationException {
      // Check that lookup methods exists.
      MethodOverrides methodOverrides = getMethodOverrides();
      if (!methodOverrides.isEmpty()) {
         for (MethodOverride mo : methodOverrides.getOverrides()) {
            prepareMethodOverride(mo);
         }
      }
   }
protected void prepareMethodOverride(MethodOverride mo) throws
BeanDefinitionValidationException {
      //获取对应类中对应方法名的个数
         int count = ClassUtils.getMethodCountForName(getBeanClass(),
mo.getMethodName());
      if (count == 0) {
         throw new BeanDefinitionValidationException(
         "Invalid method override: no method with name '" +
mo.getMethodName() +
         "' on class [" + getBeanClassName() + "]");
      }
      else if (count == 1) {
        //标记MethodOverride暂未被覆盖，避免参数类型检查的开
```

销。

```
            mo.setOverloaded(false);
        }
    }
```

通过以上两个函数的代码你能体会到它所要实现的功能吗？之前反复提到过，在Spring配置中存在lookup-method和replace-method两个配置功能，而这两个配置的加载其实就是将配置统一存放在BeanDefinition中的methodOverrides属性里，这两个功能实现原理其实是在bean实例化的时候如果检测到存在methodOverrides属性，会动态地为当前bean生成代理并使用对应的拦截器为bean做增强处理，相关逻辑实现在bean的实例化部分详细介绍。

但是，这里要提到的是，对于方法的匹配来讲，如果一个类中存在若干个重载方法，那么，在函数调用及增强的时候还需要根据参数类型进行匹配，来最终确认当前调用的到底是哪个函数。但是，Spring将一部分匹配工作在这里完成了，如果当前类中的方法只有一个，那么就设置重载该方法没有被重载，这样在后续调用的时候便可以直接使用找到的方法，而不需要进行方法的参数匹配验证了，而且还可以提前对方法存在性进行验证，正可谓一箭双雕。

5.5.2 实例化的前置处理

在真正调用doCreate方法创建bean的实例前使用了这样一个方法resolveBeforeInstantiation (beanName, mbd)对 BeanDefinigiton中的属性做些前置处理。当然，无论其中是否有相应的逻辑实现我们都可以理解，因为真正逻辑实现前后留有处理函数也是可扩展的一种体现，但是，这并不是最重要的，在函数中还提供了一个短路判断，这才是最为关键的部分。

```
if (bean != null) {
```

```
    return bean;
}
```

当经过前置处理后返回的结果如果不为空，那么会直接略过后续的Bean 的创建而直接返回结果。这一特性虽然很容易被忽略，但是却起着至关重要的作用，我们熟知的 AOP 功能就是基于这里的判断的。

```
protected Object resolveBeforeInstantiation(String beanName,
RootBeanDefinition mbd) {
    Object bean = null;
//如果尚未被解析
    if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
        // Make sure bean class is actually resolved at this point.
        if (mbd.hasBeanClass() && !mbd.isSynthetic() &&
hasInstantiationAware
        BeanPostProcessors()) {
            bean =
applyBeanPostProcessorsBeforeInstantiation(mbd.getBeanClass(),
            beanName);
                if (bean != null) {
                bean = applyBeanPostProcessorsAfterInitialization(bean,
beanName);
            }
        }
        mbd.beforeInstantiationResolved = (bean != null);
    }
    return bean;
}
```

此方法中最吸引我们的无疑是两个方法

applyBeanPostProcessorsBeforeInstantiation 以及
applyBeanPostProcessorsAfterInitialization。两个方法实现的非常简单，无非是对后处理器中的所有InstantiationAwareBeanPostProcessor类型的后处理器进行postProcessBeforeInstantiation方法和BeanPostProcessor的postProcessAfterInitialization方法的调用。

1．实例化前的后处理器应用

bean的实例化前调用，也就是将AbsractBeanDefinition转换为BeanWrapper 前的处理。给子类一个修改BeanDefinition的机会，也就是说当程序经过这个方法后，bean可能已经不是我们认为的bean了，而是或许成为了一个经过处理的代理bean，可能是通过cglib生成的，也可能是通过其它技术生成的。这在第7章中会详细介绍，我们只需要知道，在bean的实例化前会调用后处理器的方法进行处理。

```
protected Object applyBeanPostProcessorsBeforeInstantiation(Class
beanClass, String beanName)
    throws BeansException {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
            if (bp instanceof InstantiationAwareBeanPostProcessor) {
                InstantiationAwareBeanPostProcessor ibp = (Instantiation
AwareBean
                PostProcessor) bp;
                Object result = ibp.postProcessBeforeInstantiation(beanClass,
beanName);
                if (result != null) {
                    return result;
                }
            }
        }
```

```
        return null;
    }
```

2．实例化后的后处理器应用

在讲解从缓存中获取单例bean的时候就提到过，Spring中的规则是在bean的初始化后尽可能保证将注册的后处理器的postProcessAfterInitialization方法应用到该bean中，因为如果返回的bean不为空，那么便不会再次经历普通bean的创建过程，所以只能在这里应用后处理器的postProcessAfterInitialization方法。

```
    public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)
    throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
            result = beanProcessor.postProcessAfterInitialization(result, beanName);
            if (result == null) {
                return result;
            }
        }
        return result;
    }
```

5.6 循环依赖

实例化 bean 是一个非常复杂的过程，而其中最比较难以理解的就是对循环依赖的解决，不管之前读者有没有对循环依赖方面的研究，这里有必要先对此知识点稍作回顾。

5.6.1 什么是循环依赖

循环依赖就是循环引用，就是两个或多个 bean 相互之间的持有对方，比如 CircleA 引用CircleB，CircleB引用CircleC，CircleC引用CircleA，则它们最终反映为一个环。此处不是循环调用，循环调用是方法之间的环调用，如图5-2所示。

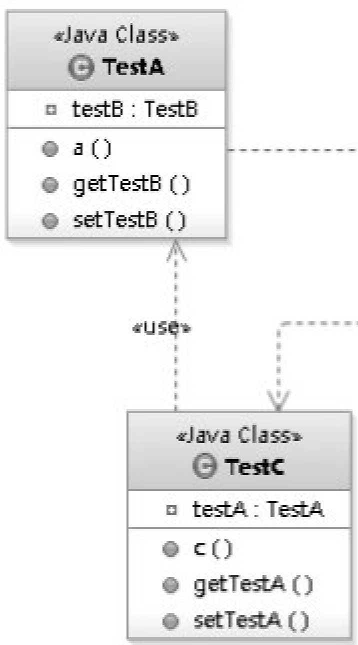


图5-2 循环依赖

循环调用是无法解决的，除非有终结条件，否则就是死循环，最终导致内存溢出错误。

5.6.2 Spring如何解决循环依赖

Spring容器循环依赖包括构造器循环依赖和setter循环依赖，那Spring容器如何解决循环依赖呢？首先让我们来定义循环引用类：

```
public class TestA {
    private TestB testB;
```

```
    public void a() {
       testB.b();
    }
    public TestB getTestB() {
        return testB;
    }
    public void setTestB(TestB testB) {
        this.testB = testB;
    }
}
public class TestB {
    private TestC testC;
    public void b() {
       testC.c();
    }
    public TestC getTestC() {
        return testC;
    }
    public void setTestC(TestC testC) {
        this.testC = testC;
    }
}
public class TestC {
    private TestA testA;
    public void c() {
       testA.a();
    }
```

```
    public TestA getTestA() {
        return testA;
    }
    public void setTestA(TestA testA) {
        this.testA = testA;
    }
}
```

在Spring中将循环依赖的处理分成了3种情况。

1．构造器循环依赖

表示通过构造器注入构成的循环依赖，此依赖是无法解决的，只能抛出 BeanCurrentlyIn CreationException异常表示循环依赖。

如在创建TestA类时，构造器需要TestB类，那将去创建TestB，在创建TestB类时又发现需要TestC类，则又去创建TestC，最终在创建TestC时发现又需要TestA，从而形成一个环，没办法创建。

Spring容器将每一个正在创建的bean标识符放在一个“当前创建bean池”中，bean标识符在创建过程中将一直保持在这个池中，因此如果在创建 bean 过程中发现自己已经在“当前创建bean池”里时，将抛出BeanCurrentlyInCreationException异常表示循环依赖；而对于创建完毕的bean将从“当前创建bean池”中清除掉。

我们通过一个直观的测试用例来进行分析。

（1）创建配置文件。

```
<bean id="testA" class="com.bean.TestA">
    <constructor-arg index="0" ref="testB"/>
</bean>
<bean id="testB" class="com.bean.TestB">
    <constructor-arg index="0" ref="testC"/>
</bean>
```

```
<bean id="testC" class="com.bean.TestC">
    <constructor-arg index="0" ref="testA"/>
</bean>
```

（2）创建测试用例。

```
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleByConstructor() throws Throwable {
    try {
        new ClassPathXmlApplicationContext("test.xml");
    } catch (Exception e) {
       //因为要在创建testC时抛出；
        Throwable e1 = e.getCause().getCause().getCause();
        throw e1;
    }
}
```

针对以上代码的分析如下。

Spring容器创建“testA”bean，首先去“当前创建bean池”查找是否当前bean正在创建，如果没发现，则继续准备其需要的构造器参数“testB”，并将“testA”标识符放到“当前创建bean池”。

Spring容器创建“testB”bean，首先去“当前创建bean池”查找是否当前bean正在创建，如果没发现，则继续准备其需要的构造器参数“testC”，并将“testB”标识符放到“当前创建bean池”。

Spring容器创建“testC”bean，首先去“当前创建bean池”查找是否当前bean正在创建，如果没发现，则继续准备其需要的构造器参数“testA”，并将“testC”标识符放到“当前创建Bean池”。

到此为止Spring容器要去创建“testA”bean，发现该bean标识符在“当前创建bean池”中，因为表示循环依赖，抛出BeanCurrentlyInCreationException。

2．setter循环依赖

表示通过setter注入方式构成的循环依赖。对于setter注入造成的依赖是通过Spring容器提前暴露刚完成构造器注入但未完成其他步骤（如setter注入）的bean来完成的，而且只能解决单例作用域的bean循环依赖。通过提前暴露一个单例工厂方法，从而使其他bean能引用到该bean，如下代码所示：

```
addSingletonFactory(beanName, new ObjectFactory() {
    public Object getObject() throws BeansException {
        return getEarlyBeanReference(beanName, mbd, bean);
    }
});
```

具体步骤如下。

（1）Spring 容器创建单例“testA”bean，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将“testA”标识符放到“当前创建bean池”，然后进行setter注入“testB”。

（2）Spring 容器创建单例“testB”bean，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将“testB”标识符放到“当前创建bean池”，然后进行setter注入“circle”。

（3）Spring 容器创建单例“testC”bean，首先根据无参构造器创建bean，并暴露一个“ObjectFactory”用于返回一个提前暴露一个创建中的bean，并将“testC”标识符放到“当前创建bean池”，然后进行setter注入“testA”。进行注入“testA”时由于提前暴露了“ObjectFactory”工厂，从而使用它返回提前暴露一个创建中的bean。

（4）最后在依赖注入“testB”和“testA”，完成setter注入。

3．prototype范围的依赖处理

对于“prototype”作用域bean，Spring容器无法完成依赖注入，因为Spring容器不进行缓存“prototype”作用域的bean，因此无法提前暴露一个创建中的bean。示例如下：

（1）创建配置文件。

```
<bean id="testA" class="com.bean.CircleA" scope="prototype">
    <property name="testB" ref="testB"/>
</bean>
<bean id="testB" class="com.bean.CircleB" scope="prototype">
    <property name="testC" ref="testC"/>
</bean>
<bean id="testC" class="com.bean.CircleC" scope="prototype">
    <property name="testA" ref="testA"/>
</bean>
```

（2）创建测试用例。

```
@Test(expected = BeanCurrentlyInCreationException.class)
public void testCircleBySetterAndPrototype () throws Throwable {
    try {
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext(
        "testPrototype.xml");
        System.out.println(ctx.getBean("testA"));
    } catch (Exception e) {
        Throwable e1 = e.getCause().getCause().getCause();
        throw e1;
    }
}
```

对于“singleton”作用域bean，可以通

过“setAllowCircularReferences(false）；”来禁用循环引用。

感谢互联网时代，让我可以方便地获取我想要的各种信息，当初我刚开始学习的时候，一直纠结于这里错综复杂的逻辑，幸好我看到了一篇文章解开了我心中的疑惑。在此，感谢原作者，并将原文与大家分享，帮助大家更好的理解Spring的依赖，大家可以从http://www.iflym.com/index.php/code/201208280001.html来获取原文。

5.7 创建bean

介绍了循环依赖以及Spring中的循环依赖的处理方式后，我们继续4.5小节的内容。当经历过 resolveBeforeInstantiation 方法后，程序有两个选择，如果创建了代理或者说重写了InstantiationAwareBeanPostProcessor 的 postProcessBeforeInstantiation 方法并在方法 postProcess BeforeInstantiation中改变了bean，则直接返回就可以了，否则需要进行常规bean的创建。而这常规bean的创建就是在doCreateBean中完成的。

```
    protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final
    Object[] args) {
        // Instantiate the bean.
        BeanWrapper instanceWrapper = null;
        if (mbd.isSingleton()) {
            instanceWrapper =
this.factoryBeanInstanceCache.remove(beanName);
        }
        if (instanceWrapper == null) {
          //根据指定bean使用对应的策略创建新的实例，如：工厂方
```

法、构造函数自动注入、简单初始化

```
                instanceWrapper = createBeanInstance(beanName, mbd, args);
            }
            final Object bean = (instanceWrapper != null ? instanceWrapper.
getWrappedInstance() : null);
            Class beanType = (instanceWrapper != null ?
instanceWrapper.getWrappedClass() : null);
            // Allow post-processors to modify the merged bean definition.
            synchronized (mbd.postProcessingLock) {
                if (!mbd.postProcessed) {
                   //应用MergedBeanDefinitionPostProcessor
                    applyMergedBeanDefinitionPostProcessors(mbd, beanType,
beanName);
                    mbd.postProcessed = true;
                }
            }
            /*
               * 是否需要提早曝光:单例&允许循环依赖&当前bean正在创建
中，检测循环依赖
               */
            boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
            isSingletonCurrentlyInCreation(beanName));
            if (earlySingletonExposure) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Eagerly caching bean '" + beanName +
                    "' to allow for resolving potential circular references");
```

```
            }
            //为避免后期循环依赖，可以在bean初始化完成前将创建实例
的ObjectFactory加入工厂
             addSingletonFactory(beanName, new ObjectFactory() {
                 public Object getObject() throws BeansException {
                     //对 bean 再一次依赖引用，主要应用
SmartInstantiationAware BeanPost
                    Processor,
                    //其中我们熟知的AOP就是在这里将advice动态织入bean
中，若没有则直接返回
                    bean，不做任何处理
                     return getEarlyBeanReference(beanName, mbd, bean);
                 }
            });
        }
        // Initialize the bean instance.
        Object exposedObject = bean;
        try {
            始依赖bean
            //对bean进行填充，将各个属性值注入，其中，可能存在依赖
于其他bean的属性，则会递归初
            populateBean(beanName, mbd, instanceWrapper);
             if (exposedObject != null) {
                //调用初始化方法，比如init-method
                 exposedObject = initializeBean(beanName, exposedObject,
mbd);
            }
```

```
            }
            catch (Throwable ex) {
                if (ex instanceof BeanCreationException &&
beanName.equals(((BeanCreationException)
                ex).getBeanName())) {
                    throw (BeanCreationException) ex;
                }
                else {
                    throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                    "Initialization of bean failed", ex);
                }
            }
            if (earlySingletonExposure) {
                Object earlySingletonReference = getSingleton(beanName, false);
               //earlySingletonReference只有在检测到有循环依赖的情况下才
会不为空
                if (earlySingletonReference != null) {
                   //如果exposedObject没有在初始化方法中被改变，也就是没
有被增强
                    if (exposedObject == bean) {
                        exposedObject = earlySingletonReference;
                    (beanName)) {
                    }else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean
                        String[] dependentBeans = getDependentBeans(beanName);
                    Set<String> actualDependentBeans = new
```

```
LinkedHashSet<String>
                    (dependentBeans.length);
                     for (String dependentBean : dependentBeans) {
                        //检测依赖
                         if
(!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                            actualDependentBeans.add(dependentBean);
                        }
                    }
                    /*
                        * 因为bean创建后其所依赖的bean一定是已经创建的，
                        * actualDependentBeans不为空则表示当前bean创建后其依
赖的bean却没有
                        没全部创建完，也就是说存在循环依赖
                        */
                     if (!actualDependentBeans.isEmpty()) {
                         throw new BeanCurrentlyInCreationException(beanName,
                         "Bean with name '" + beanName + "' has been injected
                         into other beans [" +
                         (actualDependentBeans) +
                        StringUtils.collectionToCommaDelimitedString
                         "] in its raw version as part of a circular reference,
                         but has eventually been " +
                         use the final version of the " +
                         "wrapped. This means that said other beans do not
                         "bean. This is often the result of over-eager type
                         matching - consider using " +
```

```
                flag turned off, for example.");
                "'getBeanNamesOfType' with the 'allowEagerInit'
               }
            }
         }
      }
      // Register bean as disposable.
      try {
        //根据scopse注册bean
        registerDisposableBeanIfNecessary(beanName, bean, mbd);
      }
      catch (BeanDefinitionValidationException ex) {
         throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
         "Invalid destruction signature", ex);
      }
      return exposedObject;
   }
```

尽管日志与异常的内容非常重要，但是在阅读源码的时候似乎大部分人都会直接忽略掉。在此不深入探讨日志及异常的设计，我们看看整个函数的概要思路。

（1）如果是单例则需要首先清除缓存。

（2）实例化bean，将BeanDefinition转换为BeanWrapper。

转换是一个复杂的过程，但是我们可以尝试概括大致的功能，如下所示。

如果存在工厂方法则使用工厂方法进行初始化。

一个类有多个构造函数，每个构造函数都有不同的参数，所以需要

根据参数锁定构造函数并进行初始化。

如果既不存在工厂方法也不存在带有参数的构造函数，则使用默认的构造函数进行bean的实例化。

（3）MergedBeanDefinitionPostProcessor的应用。

bean合并后的处理，Autowired注解正是通过此方法实现诸如类型的预解析。

（4）依赖处理。

在Spring中会有循环依赖的情况，例如，当A中含有B的属性，而B中又含有A的属性时就会构成一个循环依赖，此时如果A和B都是单例，那么在Spring中的处理方式就是当创建B的时候，涉及自动注入A的步骤时，并不是直接去再次创建A，而是通过放入缓存中的ObjectFactory来创建实例，这样就解决了循环依赖的问题。

（5）属性填充。将所有属性填充至bean的实例中。

（6）循环依赖检查。

之前有提到过，在Sping中解决循环依赖只对单例有效，而对于prototype的bean，Spring没有好的解决办法，唯一要做的就是抛出异常。在这个步骤里面会检测已经加载的 bean 是否已经出现了依赖循环，并判断是否需要抛出异常。

（7）注册DisposableBean。

如果配置了destroy-method，这里需要注册以便于在销毁时候调用。

（8）完成创建并返回。

可以看到上面的步骤非常的繁琐，每一步骤都使用了大量的代码来完成其功能，最复杂也是最难以理解的当属循环依赖的处理，在真正进入doCreateBean前我们有必要先了解下循环依赖。

5.7.1 创建bean的实例

当我们了解了循环依赖以后就可以深入分析创建 bean 的每个步骤了，首先我们从createBeanInstance开始。

```
protected BeanWrapper createBeanInstance(String beanName,
RootBeanDefinition mbd, Object[] args) {
    //解析class
    Class beanClass = resolveBeanClass(mbd, beanName);
    if (beanClass != null &&
!Modifier.isPublic(beanClass.getModifiers()) && !mbd.
    isNonPublicAccessAllowed()) {
        throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
        "Bean class isn't public, and non-public access not allowed: " +
        beanClass.getName());
    }
    //如果工厂方法不为空则使用工厂方法初始化策略
    if (mbd.getFactoryMethodName() != null) {
        return instantiateUsingFactoryMethod(beanName, mbd, args);
    }
    // Shortcut when re-creating the same bean...
    boolean resolved = false;
    boolean autowireNecessary = false;
    if (args == null) {
        synchronized (mbd.constructorArgumentLock) {
            //一个类有多个构造函数，每个构造函数都有不同的参数，
所以调用前需要先根据参数锁定
            构造函数或对应的工厂方法
            if (mbd.resolvedConstructorOrFactoryMethod != null) {
```

```
                resolved = true;
                autowireNecessary = mbd.constructorArgumentsResolved;
            }
        }
    }
    //如果已经解析过则使用解析好的构造函数方法不需要再次锁定
    if (resolved) {
        if (autowireNecessary) {
          //构造函数自动注入
          return autowireConstructor(beanName, mbd, null, null);
        }
        else {
          //使用默认构造函数构造
          return instantiateBean(beanName, mbd);
        }
    }
    //需要根据参数解析构造函数
    Constructor[] ctors =
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
    if (ctors != null ||
          mbd.getResolvedAutowireMode() == RootBeanDefinition.
AUTOWIRE_
          CONSTRUCTOR ||
          mbd.hasConstructorArgumentValues() ||
!ObjectUtils.isEmpty(args)) {
        //构造函数自动注入
        return autowireConstructor(beanName, mbd, ctors, args);
```

```
    }
    //使用默认构造函数构造
     return instantiateBean(beanName, mbd);
}
```

虽然代码中实例化的细节非常复杂，但是在createBeanIntance方法中我们还是可以清晰地看到实例化的逻辑的。

（1）如果在RootBeanDefinition中存在factoryMethodName属性，或者说在配置文件中配置了factory-method，那么Spring会尝试使用instantiateUsingFactoryMethod(beanName, mbd, args)方法根据RootBeanDefinition中的配置生成bean的实例。

（2）解析构造函数并进行构造函数的实例化。因为一个 bean 对应的类中可能会有多个构造函数，而每个构造函数的参数不同，Spring在根据参数及类型去判断最终会使用哪个构造函数进行实例化。但是，判断的过程是个比较消耗性能的步骤，所以采用缓存机制，如果已经解析过则不需要重复解析而是直接从RootBeanDefinition中的属性resolvedConstructorOrFactoryMethod缓存的值去取，否则需要再次解析，并将解析的结果添加至 RootBeanDefinition 中的属性resolvedConstructorOrFactoryMethod中。

1．autowireConstructor

对于实例的创建Spring中分成了两种情况，一种是通用的实例化，另一种是带有参数的实例化。带有参数的实例化过程相当复杂，因为存在着不确定性，所以在判断对应参数上做了大量工作。

```
public BeanWrapper autowireConstructor(
final String beanName, final RootBeanDefinition mbd, Constructor[]
chosenCtors, final Object[] explicitArgs) {
     BeanWrapperImpl bw = new BeanWrapperImpl();
    this.beanFactory.initBeanWrapper(bw);
```

```
Constructor constructorToUse = null;
ArgumentsHolder argsHolderToUse = null;
Object[] argsToUse = null;
//explicitArgs通过getBean方法传入
//如果getBean方法调用的时候指定方法参数那么直接使用
if (explicitArgs != null) {
    argsToUse = explicitArgs;
}else {
    //如果在getBean方法时候没有指定则尝试从配置文件中解析
    Object[] argsToResolve = null;
    //尝试从缓存中获取
    synchronized (mbd.constructorArgumentLock) {
        constructorToUse = (Constructor)
mbd.resolvedConstructorOrFactoryMethod;
        if (constructorToUse != null &&
mbd.constructorArgumentsResolved) {
            //从缓存中取
            argsToUse = mbd.resolvedConstructorArguments;
            if (argsToUse == null) {
                //配置的构造函数参数
                argsToResolve = mbd.preparedConstructorArguments;
            }
        }
    }
    //如果缓存中存在
    if (argsToResolve != null) {
        //解析参数类型，如给定方法的构造函数 A(int,int)则通过此
```

```
方法后就会把配置中的
            ("1","1")转换为（1,1）
            //缓存中的值可能是原始值也肯能是最终值
             argsToUse = resolvePreparedArguments(beanName, mbd, bw,
constructorToUse,
            argsToResolve);
          }
        }
        //没有被缓存
         if (constructorToUse == null) {
           // Need to resolve the constructor.
           boolean autowiring = (chosenCtors != null ||
             mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_
            CONSTRUCTOR);
           ConstructorArgumentValues resolvedValues = null;
           int minNrOfArgs;
           if (explicitArgs != null) {
             minNrOfArgs = explicitArgs.length;
           }else {
            //提取配置文件中的配置的构造函数参数
             ConstructorArgumentValues cargs =
mbd.getConstructorArgumentValues();
            //用于承载解析后的构造函数参数的值
             resolvedValues = new ConstructorArgumentValues();
            //能解析到的参数个数
             minNrOfArgs = resolveConstructorArguments(beanName, mbd,
```

```
bw, cargs,
                    resolvedValues);
                }
                // Take specified constructors, if any.
                Constructor[] candidates = chosenCtors;
                if (candidates == null) {
                    Class beanClass = mbd.getBeanClass();
                    try {
                        candidates = (mbd.isNonPublicAccessAllowed() ?
                       beanClass.getDeclaredConstructors():beanClass.
                       getConstructors());
                    }
                    catch (Throwable ex) {
                        throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                        "Resolution of declared constructors on bean Class [" +
                        beanClass.getName() +
                        "] from ClassLoader [" + beanClass. getClassLoader()
                        + "] failed", ex);
                    }
                }
                //排序给定的构造函数，public构造函数优先参数数量降序、非
public构造函数参数数量降序
                AutowireUtils.sortConstructors(candidates);
                int minTypeDiffWeight = Integer.MAX_VALUE;
                Set<Constructor> ambiguousConstructors = null;
                List<Exception> causes = null;
```

```
for (int i = 0; i < candidates.length; i++) {
    Constructor<?> candidate = candidates[i];
    Class[] paramTypes = candidate.getParameterTypes();
    if (constructorToUse != null && argsToUse.length >
paramTypes.length) {
        //如果已经找到选用的构造函数或者需要的参数个数小于
当前的构造函数参数个数则
        终止,因为已经按照参数个数降序排列
        break;
    }
    if (paramTypes.length < minNrOfArgs) {
        //参数个数不相等
        continue;
    }
    ArgumentsHolder argsHolder;
    if (resolvedValues != null) {
        //有参数则根据值构造对应参数类型的参数
        try {
            String[] paramNames = null;
            if (constructorPropertiesAnnotationAvailable) {
                //注释上获取参数名称
                (candidate, paramTypes.length);
                paramNames = ConstructorPropertiesChecker.
evaluateAnnotation
            }
            if (paramNames == null) {
                //获取参数名称探索器
```

```
NameDiscoverer();
 ParameterNameDiscoverer pnd = this.beanFactory.
getParameter
 if (pnd != null) {
   //获取指定构造函数的参数名称
    paramNames = pnd.getParameterNames(candidate);
 }
}
//根据名称和数据类型创建参数持有者
argsHolder = createArgumentArray(
beanName, mbd, resolvedValues, bw, paramTypes,
paramNames,
candidate, autowiring);
}
catch (UnsatisfiedDependencyException ex) {
   if (this.beanFactory.logger.isTraceEnabled()) {
      this.beanFactory.logger.trace(
       "Ignoring constructor [" + candidate + "] of
       bean '" + beanName + "': " + ex);
   }
   if (i == candidates.length - 1 && constructorToUse ==
null) {
       if (causes != null) {
          for (Exception cause : causes) {
             this.beanFactory.onSuppressedException(cause);
          }
       }
```

```
                throw ex;
            }
            else {
                // Swallow and try next constructor.
                if (causes == null) {
                    causes = new LinkedList<Exception>();
                }
                causes.add(ex);
                continue;
            }
        }
    }else {
        if (paramTypes.length != explicitArgs.length) {
            continue;
        }
        //构造函数没有参数的情况
    argsHolder = new ArgumentsHolder(explicitArgs);
}
//探测是否有不确定性的构造函数存在，例如不同构造函数
的参数为父子关系
 int typeDiffWeight = (mbd.isLenientConstructorResolution() ?
    argsHolder.getTypeDifferenceWeight(paramTypes) :
argsHolder.
   getAssignabilityWeight(paramTypes));
//如果它代表着当前最接近的匹配则选择作为构造函数
 if (typeDiffWeight < minTypeDiffWeight) {
    constructorToUse = candidate;
```

```
                    argsHolderToUse = argsHolder;
                    argsToUse = argsHolder.arguments;
                    minTypeDiffWeight = typeDiffWeight;
                    ambiguousConstructors = null;
                }else if (constructorToUse != null && typeDiffWeight ==
minTypeDiffWeight) {
                    if (ambiguousConstructors == null) {
                        ambiguousConstructors = new
LinkedHashSet<Constructor>();
                       ambiguousConstructors.add(constructorToUse);
                    }
                   ambiguousConstructors.add(candidate);
                }
            }
            if (constructorToUse == null) {
                throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                "Could not resolve matching constructor " +
                "(hint: specify index/type/name arguments for simple
                parameters to avoid type ambiguities)");
            }else if (ambiguousConstructors != null &&
!mbd.isLenientConstructor
            Resolution()) {
                throw new
BeanCreationException(mbd.getResourceDescription(), beanName,
                "Ambiguous constructor matches found in bean '" + beanName
+ "' " +
```

```
                "(hint: specify index/type/name arguments for simple
                parameters to avoid type ambiguities): " +
               ambiguousConstructors);
            }
            if (explicitArgs == null) {
               //将解析的构造函数加入缓存
                argsHolderToUse.storeCache(mbd, constructorToUse);
            }
        }
        try {
            Object beanInstance;
            if (System.getSecurityManager() != null) {
                final Constructor ctorToUse = constructorToUse;
                final Object[] argumentsToUse = argsToUse;
                beanInstance = AccessController.doPrivileged(new
PrivilegedAction<
                Object>() {
                    public Object run() {
                        return beanFactory.getInstantiationStrategy().instantiate(
                        mbd, beanName, beanFactory, ctorToUse,
argumentsToUse);
                    }
                }, beanFactory.getAccessControlContext());
            }
            else {
                beanInstance =
this.beanFactory.getInstantiationStrategy().instantiate(
```

```
                    mbd, beanName, this.beanFactory, constructorToUse,
argsToUse);
            }
            //将构建的实例加入BeanWrapper中
            bw.setWrappedInstance(beanInstance);
             return bw;
        }
         catch (Throwable ex) {
             throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
             "Instantiation of bean failed", ex);
        }
    }
```

逻辑很复杂，函数代码量很大，不知道你是否坚持读完了整个函数并理解了整个功能呢？这里要先吐个槽，笔者觉得这个函数的写法完全不符合 Spring 的一贯风格，如果你一直跟随笔者的分析思路到这里，相信你或多或少对Spring的编码风格有所了解，Spring的一贯做法是将复杂的逻辑分解，分成N个小函数的嵌套，每一层都是对下一层逻辑的总结及概要，这样使得每一层的逻辑会变得简单容易理解。在上面的函数中，包含着很多的逻辑实现，笔者觉得至少应该将逻辑封装在不同函数中而使得在autowireConstructor中的逻辑清晰明了。

我们总览一下整个函数，其实现的功能考虑了以下几个方面。

（1）构造函数参数的确定。

根据explicitArgs参数判断。

如果传入的参数explicitArgs不为空，那边可以直接确定参数，因为explicitArgs参数是在调用Bean的时候用户指定的，在BeanFactory类中存在这样的方法：

Object getBean(String name, Object... args) throws BeansException;

在获取bean的时候，用户不但可以指定bean的名称还可以指定bean所对应类的构造函数或者工厂方法的方法参数，主要用于静态工厂方法的调用，而这里是需要给定完全匹配的参数的，所以，便可以判断，如果传入参数explicitArgs不为空，则可以确定构造函数参数就是它。

缓存中获取。

除此之外，确定参数的办法如果之前已经分析过，也就是说构造函数参数已经记录在缓存中，那么便可以直接拿来使用。而且，这里要提到的是，在缓存中缓存的可能是参数的最终类型也可能是参数的初始类型，例如：构造函数参数要求的是 int 类型，但是原始的参数值可能是String类型的“1”，那么即使在缓存中得到了参数，也需要经过类型转换器的过滤以确保参数类型与对应的构造函数参数类型完全对应。

配置文件获取。

如果不能根据传入的参数explicitArgs确定构造函数的参数也无法在缓存中得到相关信息，那么只能开始新一轮的分析了。

分析从获取配置文件中配置的构造函数信息开始，经过之前的分析，我们知道，Spring中配置文件中的信息经过转换都会通过BeanDefinition实例承载，也就是参数mbd中包含，那么可以通过调用mbd.getConstructorArgumentValues()来获取配置的构造函数信息。有了配置中的信息便可以获取对应的参数值信息了，获取参数值的信息包括直接指定值，如：直接指定构造函数中某个值为原始类型 String 类型，或者是一个对其他 bean 的引用，而这一处理委托给resolveConstructorArguments方法，并返回能解析到的参数的个数。

（2）构造函数的确定。

经过了第一步后已经确定了构造函数的参数，接下来的任务就是根据构造函数参数在所有构造函数中锁定对应的构造函数，而匹配的方法就是根据参数个数匹配，所以在匹配之前需要先对构造函数按照public

构造函数优先参数数量降序、非public构造函数参数数量降序。这样可以在遍历的情况下迅速判断排在后面的构造函数参数个数是否符合条件。

由于在配置文件中并不是唯一限制使用参数位置索引的方式去创建，同样还支持指定参数名称进行设定参数值的情况，如<constructor-arg name="aa">，那么这种情况就需要首先确定构造函数中的参数名称。

获取参数名称可以有两种方式，一种是通过注解的方式直接获取，另一种就是使用Spring中提供的工具类 ParameterNameDiscoverer 来获取。构造函数、参数名称、参数类型、参数值都确定后就可以锁定构造函数以及转换对应的参数类型了。

（3）根据确定的构造函数转换对应的参数类型。

主要是使用Spring中提供的类型转换器或者用户提供的自定义类型转换器进行转换。

（4）构造函数不确定性的验证。

当然，有时候即使构造函数、参数名称、参数类型、参数值都确定后也不一定会直接锁定构造函数，不同构造函数的参数为父子关系，所以Spring在最后又做了一次验证。

（5）根据实例化策略以及得到的构造函数及构造函数参数实例化Bean。后面章节中将进行讲解。

2．instantiateBean

经历了带有参数的构造函数的实例构造，相信你会非常轻松愉快地理解不带参数的构造函数的实例化过程。

```
protected BeanWrapper instantiateBean(final String beanName, final RootBean
Definition mbd) {
    try {
```

```
                Object beanInstance;
                final BeanFactory parent = this;
                if (System.getSecurityManager() != null) {
                    beanInstance = AccessController.doPrivileged(new
PrivilegedAction
                    <Object>() {
                        public Object run() {
                            return getInstantiationStrategy().instantiate(mbd,
beanName,
                           parent);
                        }
                    }, getAccessControlContext());
                }
                else {
                    beanInstance = getInstantiationStrategy().instantiate(mbd,
beanName,
                   parent);
                }
                BeanWrapper bw = new BeanWrapperImpl(beanInstance);
               initBeanWrapper(bw);
                return bw;
            }
            catch (Throwable ex) {
                throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
                "Instantiation of bean failed", ex);
            }
```

}

你会发现，此方法并没有什么实质性的逻辑，带有参数的实例构造中，Spring把精力都放在了构造函数以及参数的匹配上，所以如果没有参数的话那将是非常简单的一件事，直接调用实例化策略进行实例化就可以了。

3．实例化策略

实例化过程中反复提到过实例化策略，那这又是做什么用的呢？其实，经过前面的分析，我们已经得到了足以实例化的所有相关信息，完全可以使用最简单的反射方法直接反射来构造实例对象，但是Spring却并没有这么做。

SimpleInstantiationStrategy.java

```
public Object instantiate(RootBeanDefinition beanDefinition, String
beanName, BeanFactory
owner) {
  //如果有需要覆盖或者动态替换的方法则当然需要使用 cglib 进行
动态代理，因为可以在创建代理的同
  时将动态方法织入类中，
  //但是如果没有需要动态改变得方法，为了方便直接反射就可以
了
   if (beanDefinition.getMethodOverrides().isEmpty()) {
      Constructor<?> constructorToUse;
      synchronized (beanDefinition.constructorArgumentLock) {
         constructorToUse = (Constructor<?
>)beanDefinition.resolvedConstructor
        OrFactoryMethod;
         if (constructorToUse == null) {
            final Class clazz = beanDefinition.getBeanClass();
```

```
if (clazz.isInterface()) {
    throw new BeanInstantiationException(clazz, "Specified
class
    is an interface");
}
try {
    if (System.getSecurityManager() != null) {
        constructorToUse = AccessController.doPrivileged(new
        PrivilegedExceptionAction<Constructor>() {
            public Constructor run() throws Exception {
                return clazz.getDeclaredConstructor((Class[]) null);
            }
        });
    }
    else {
       null);
        constructorToUse =
clazz.getDeclaredConstructor((Class[])
    }
   constructorToUse;
    beanDefinition.resolvedConstructorOrFactoryMethod =
}
catch (Exception ex) {
    constructor found", ex);
    throw new BeanInstantiationException(clazz, "No default
}
}
```

```
            }
            return BeanUtils.instantiateClass(constructorToUse);
        }else {
            // Must generate CGLIB subclass.
            return instantiateWithMethodInjection(beanDefinition, beanName, owner);
        }
    }
```

CglibSubclassingInstantiationStrategy.java

```
public Object instantiate(Constructor ctor, Object[] args) {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(this.beanDefinition.getBeanClass());
    enhancer.setCallbackFilter(new CallbackFilterImpl());
    enhancer.setCallbacks(new Callback[] {
        NoOp.INSTANCE,
        new LookupOverrideMethodInterceptor(),
        new ReplaceOverrideMethodInterceptor()
    });
    return (ctor == null) ?
        enhancer.create() :
        enhancer.create(ctor.getParameterTypes(), args);
}
```

看了上面两个函数后似乎我们已经感受到了 Spring 的良苦用心以及为了能更方便地使用Spring而做了大量的工作。程序中，首先判断如果beanDefinition.getMethodOverrides()为空也就是用户没有使用replace或者lookup的配置方法，那么直接使用反射的方式，简单快捷，但是如果使用了这两个特性，在直接使用反射的方式创建实例就不妥了，因为需要

将这两个配置提供的功能切入进去，所以就必须要使用动态代理的方式将包含两个特性所对应的逻辑的拦截增强器设置进去，这样才可以保证在调用方法的时候会被相应的拦截器增强，返回值为包含拦截器的代理实例。

对于拦截器的处理方法非常简单，不再详细介绍，如果读者有兴趣，可以仔细研读第7章中关于AOP的介绍，对动态代理方面的知识会有更详细地介绍。

5.7.2 记录创建bean的ObjectFactory

在doCreate函数中有这样一段代码：

```
boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
isSingletonCurrentlyInCreation(beanName));
    if (earlySingletonExposure) {
        if (logger.isDebugEnabled()) {
            logger.debug("Eagerly caching bean '" + beanName +
            "' to allow for resolving potential circular references");
        }
        //为避免后期循环依赖，可以在bean初始化完成前将创建实例的ObjectFactory加入工厂
        addSingletonFactory(beanName, new ObjectFactory() {
            public Object getObject() throws BeansException {
                Processor,
                 //对 bean 再一次依赖引用，主要应用SmartInstantiationAware BeanPost
                bean，不做任何处理
                //其中我们熟知的AOP就是在这里将advice动态织入bean
```

中，若没有则直接返回

```
                return getEarlyBeanReference(beanName, mbd, bean);
            }
        });
    }
```

这段代码不是很复杂，但是很多人不是太理解这段代码的作用，而且，这段代码仅从此函数中去理解也很难弄懂其中的含义，我们需要从全局的角度去思考Spring的依赖解决办法。

earlySingletonExposure：从字面的意思理解就是提早曝光的单例，我们暂不定义它的学名叫什么，我们感兴趣的是有哪些条件影响这个值。

mbd.isSingleton()：没有太多可以解释的，此RootBeanDefinition代表的是否是单例。

this.allowCircularReferences：是否允许循环依赖，很抱歉，并没有找到在配置文件中如何配置，但是在AbstractRefreshableApplicationContext 中提供了设置函数，可以通过硬编码的方式进行设置或者可以通过自定义命名空间进行配置，其中硬编码的方式代码如下。

```
ClassPathXmlApplicationContext bf = new
ClassPathXmlApplicationContext ("aspectTest.xml");
    bf.setAllowBeanDefinitionOverriding(false);
```

isSingletonCurrentlyInCreation(beanName)：该bean是否在创建中。在Spring中，会有个专门的属性默认为DefaultSingletonBeanRegistry的singletonsCurrentlyInCreation来记录bean的加载状态，在bean开始创建前会将beanName记录在属性中，在bean创建结束后会将 beanName 从属性中移除。那么我们跟随代码一路走来可是对这个属性的记录并没有多少印象，这个状态是在哪里记录的呢？不同scope的记录位置并不一样，

我们以singleton为例，在singleton下记录属性的函数是在DefaultSingletonBeanRegistry类的 public Object getSingleton(String beanName, ObjectFactory singletonFactory)函数的beforeSingletonCreation(beanName)和 afterSingletonCreation(beanName)中，在这两段函数中分别this.singletonsCurrentlyInCreation.add(beanName)与this.singletonsCurrentlyIn Creation.remove(beanName)来进行状态的记录与移除。

经过以上分析我们了解变量earlySingletonExposure是否是单例、是否允许循环依赖、是否对应的bean正在创建的条件的综合。当这3个条件都满足时会执行addSingletonFactory操作，那么加入SingletonFactory的作用是什么呢？又是在什么时候调用呢？

我们还是以最简单AB循环依赖为例，类A中含有属性类B，而类B中又会含有属性类A，那么初始化beanA的过程如图5-3所示。

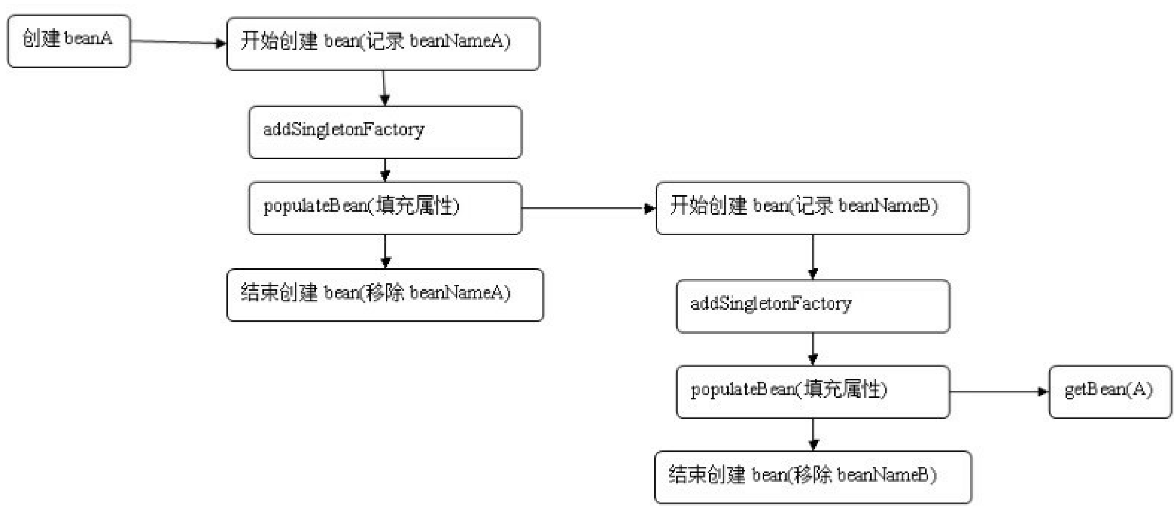


图5-3 处理循环依赖

图5-3中展示了创建beanA的流程，图中我们看到，在创建A的时候

首先会记录类A所对应的beanName，并将beanA的创建工厂加入缓存中，而在对A的属性填充也就是调用populate方法的时候又会再一次的对B进行递归创建。同样的，因为在B中同样存在A属性，因此在实例化B的的populate方法中又会再次地初始化B，也就是图形的最后，调用getBean(A)。关键是在这里，有心的同学可以去找找这个代码的实现方式，我们之前已经讲过，在这个函数中并不是直接去实例化A，而是先去检测缓存中是否有已经创建好的对应的bean，或者是否已经创建好的ObjectFactory，而此时对于A的ObjectFactory我们早已经创建，所以便不会再去向后执行，而是直接调用ObjectFactory去创建A。这里最关键的是ObjectFactory的实现。

```
addSingletonFactory(beanName, new ObjectFactory() {
    public Object getObject() throws BeansException {
        //对 bean 再一次依赖引用，主要应用 SmartInstantiationAware
BeanPost
        Processor,
        bean，不做任何处理
        //其中我们熟知的AOP就是在这里将advice动态织入bean中，若
没有则直接返回
        return getEarlyBeanReference(beanName, mbd, bean);
    }
});
```

其中getEarlyBeanReference的代码如下：

```
protected Object getEarlyBeanReference(String beanName,
RootBeanDefinition mbd, Object
bean) {
    Object exposedObject = bean;
    if (bean != null && !mbd.isSynthetic() &&
```

```
hasInstantiationAwareBean PostProcessors()) {
            for (BeanPostProcessor bp : getBeanPostProcessors()) {
               if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {
                  SmartInstantiationAwareBeanPostProcessor ibp =
(SmartInstantiation
                  AwareBeanPostProcessor) bp;
                  exposedObject = ibp.getEarlyBeanReference(exposedObject,
beanName);
                  if (exposedObject == null) {
                     return exposedObject;
                  }
               }
            }
         }
         return exposedObject;
      }
```

在getEarlyBeanReference函数中并没有太多的逻辑处理，或者说除了后处理器的调用外没有别的处理工作，根据以上分析，基本可以理清Spring 处理循环依赖的解决办法，在 B 中创建依赖A时通过ObjectFactory提供的实例化方法来中断A中的属性填充，使B中持有的A仅仅是刚刚初始化并没有填充任何属性的A，而这正初始化A的步骤还是在最开始创建A的时候进行的，但是因为A与B中的A所表示的属性地址是一样的，所以在A中创建好的属性填充自然可以通过B中的A获取，这样就解决了循环依赖的问题。

5.7.3 属性注入

在了解循环依赖的时候，我们曾经反复提到了populateBean这个函

数，也多少了解了这个函数的主要功能就是属性填充，那么究竟是如何实现填充的呢？

protected void populateBean(String beanName, AbstractBeanDefinition mbd, BeanWrapper bw) {

PropertyValues pvs = mbd.getPropertyValues();

if (bw == null) {

if (!pvs.isEmpty()) {

throw new BeanCreationException(

mbd.getResourceDescription(), beanName, "Cannot apply property

values to null instance");

}

else {

//没有可填充的属性

return;

}

}

//给InstantiationAwareBeanPostProcessors最后一次机会在属性设置前来改变bean

//如：可以用来支持属性注入的类型

boolean continueWithPropertyPopulation = true;

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

PostProcessor) bp;

InstantiationAwareBeanPostProcessor ibp = (Instantiation

```
AwareBean
                    //返回值为是否继续填充bean
                     if
(!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(),
                    beanName)) {
                     continueWithPropertyPopulation = false;
                    break;
                 }
              }
           }
       }
       //如果后处理器发出停止填充命令则终止后续的执行
       if (!continueWithPropertyPopulation) {
          return;
       }
       if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME ||
       mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
           MutablePropertyValues newPvs = new MutablePropertyValues(pvs);
           // Add property values based on autowire by name if applicable.
          //根据名称自动注入
           if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_NAME) {
              autowireByName(beanName, mbd, bw, newPvs);
           }
           // Add property values based on autowire by type if applicable.
```

```
            //根据类型自动注入
            if (mbd.getResolvedAutowireMode() ==
RootBeanDefinition.AUTOWIRE_BY_TYPE) {
                autowireByType(beanName, mbd, bw, newPvs);
            }
            pvs = newPvs;
        }
        //后处理器已经初始化
        boolean hasInstAwareBpps =
hasInstantiationAwareBeanPostProcessors();
        //需要依赖检查
        boolean needsDepCheck = (mbd.getDependencyCheck() !=
RootBeanDefinition.
        DEPENDENCY_CHECK_NONE);
        if (hasInstAwareBpps || needsDepCheck) {
            PropertyDescriptor[] filteredPds = filterPropertyDescriptors
ForDependency
        Check(bw);
            if (hasInstAwareBpps) {
                for (BeanPostProcessor bp : getBeanPostProcessors()) {
                    if (bp instanceof InstantiationAwareBeanPostProcessor) {
                        InstantiationAwareBeanPostProcessor ibp =
(InstantiationAwareBean
                        PostProcessor) bp;
                        //对所有需要依赖检查的属性进行后处理
                        pvs = ibp.postProcessPropertyValues(pvs, filteredPds,
                        bw.getWrappedInstance(), beanName);
```

```
                    if (pvs == null) {
                        return;
                    }
                }
            }
        }
            if (needsDepCheck) {
                //依赖检查，对应depends-on属性，3.0已经弃用此属性
                 checkDependencies(beanName, mbd, filteredPds, pvs);
            }
        }
        //将属性应用到bean中
        applyPropertyValues(beanName, mbd, bw, pvs);
    }
```

在populateBean函数中提供了这样的处理流程。

（1）InstantiationAwareBeanPostProcessor处理器的postProcessAfterInstantiation函数的应用，此函数可以控制程序是否继续进行属性填充。

（2）根据注入类型（byName/byType），提取依赖的bean，并统一存入PropertyValues中。

（3）应用InstantiationAwareBeanPostProcessor处理器的postProcessPropertyValues方法，对属性获取完毕填充前对属性的再次处理，典型应用是 RequiredAnnotationBeanPostProcessor 类中对属性的验证。

（4）将所有PropertyValues中的属性填充至BeanWrapper中。

在上面的步骤中有几个地方是我们比较感兴趣的，它们分别是依赖注入（ autowire ByName/autowireByType）以及属性填充，那么，接下

来进一步分析这几个功能的实现细节。

1．autowireByName

上文提到根据注入类型（byName/byType），提取依赖的bean，并统一存入PropertyValues中，那么我们首先了解下byName功能是如何实现的。

```
protected void autowireByName(
String beanName, AbstractBeanDefinition mbd, BeanWrapper bw, MutableProperty
Values pvs) {
  //寻找bw中需要依赖注入的属性
    String[] propertyNames = unsatisfiedNonSimpleProperties(mbd, bw);
    for (String propertyName : propertyNames) {
        if (containsBean(propertyName)) {
          //递归初始化相关的bean
           Object bean = getBean(propertyName);
           pvs.add(propertyName, bean);
          //注册依赖
           registerDependentBean(propertyName, beanName);
           if (logger.isDebugEnabled()) {
               logger.debug("Added autowiring by name from bean name '" + beanName +
               "' via property '" + propertyName + "' to bean named '"
               + propertyName + "'");
           }
        }
        else {
```

```
                    if (logger.isTraceEnabled()) {
                        logger.trace("Not autowiring property '" + propertyName + "'
of
                        bean '" + beanName +
                        "' by name: no matching bean found");
                    }
                }
            }
        }
```

如果读者之前了解了autowire的使用方法，相信理解这个函数的功能不会太困难，无非是在传入的参数pvs中找出已经加载的bean，并递归实例化，进而加入到pvs中。

2．autowireByType

autowireByType 与 autowireByName 对于我们理解与使用来说复杂程度都很相似，但是其实现功能的复杂度却完全不一样。

```
        protected void autowireByType(
        String beanName, AbstractBeanDefinition mbd, BeanWrapper bw,
MutableProperty Values pvs) {
            TypeConverter converter = getCustomTypeConverter();
            if (converter == null) {
                converter = bw;
            }
            Set<String> autowiredBeanNames = new LinkedHashSet<String>
(4);
           //寻找bw中需要依赖注入的属性
            String[] propertyNames = unsatisfiedNonSimpleProperties(mbd,
bw);
```

```
for (String propertyName : propertyNames) {
    try {
        PropertyDescriptor pd =
bw.getPropertyDescriptor(propertyName);
        // Don't try autowiring by type for type Object: never makes
sense,
        // even if it technically is a unsatisfied, non-simple property.
        if (!Object.class.equals(pd.getPropertyType())) {
          //探测指定属性的set方法
           MethodParameter methodParam = BeanUtils.getWriteMethod
Parameter(pd);
           boolean eager = !PriorityOrdered.class. isAssignableFrom
(bw.get
          WrappedClass());
           DependencyDescriptor desc = new
AutowireByTypeDependencyDescriptor
           (methodParam, eager);
          //解析指定 beanName 的属性所匹配的值，并把解析到的
属性名称存储在
          autowiredBeanNames中，当属性存在多个封装bean时如：
           //@Autowired private List<A> aList; 将会找到所有匹配A类
型的bean并
          将其注入
           Object autowiredArgument = resolveDependency(desc,
beanName,
           autowiredBeanNames, converter);
           if (autowiredArgument != null) {
```

```
                    pvs.add(propertyName, autowiredArgument);
                }
                for (String autowiredBeanName : autowiredBeanNames) {
                  //注册依赖
                    registerDependentBean(autowiredBeanName, beanName);
                    if (logger.isDebugEnabled()) {
                        beanName + "' via property '" +
                        logger.debug("Autowiring by type from bean name '" +
                        propertyName + "' to bean named '" +
                        autowiredBeanName + "'");
                    }
                }
                autowiredBeanNames.clear();
            }
        }
        catch (BeansException ex) {
            throw new
UnsatisfiedDependencyException(mbd.getResourceDescription(),
            beanName, propertyName, ex);
        }
    }
}
```

实现根据名称自动匹配的第一步就是寻找bw中需要依赖注入的属性，同样对于根据类型自动匹配的实现来讲第一步也是寻找bw中需要依赖注入的属性，然后遍历这些属性并寻找类型匹配的bean，其中最复杂的就是寻找类型匹配的bean。同时，Spring中提供了对集合的类型注入的支持，如使用注解的方式：

```
@Autowired
    private List<Test> tests;
```

Spring 将会把所有与Test匹配的类型找出来并注入到tests属性中，正是由于这一因素，所以在 autowireByType 函数中，新建了局部遍历 autowiredBeanNames，用于存储所有依赖的bean，如果只是对非集合类的属性注入来说，此属性并无用处。

对于寻找类型匹配的逻辑实现封装在了resolveDependency函数中。

DefaultListableBeanFactory.java

```
public Object resolveDependency(DependencyDescriptor descriptor,
String beanName,
Set<String> autowiredBeanNames, TypeConverter typeConverter)
throws
BeansException {

descriptor.initParameterNameDiscovery(getParameterNameDiscoverer());
        if (descriptor.getDependencyType().equals(ObjectFactory.class)) {
            //ObjectFactory类注入的特殊处理
             return new DependencyObjectFactory(descriptor, beanName);
        }
        else if
(descriptor.getDependencyType().equals(javaxInjectProviderClass)) {
            //javaxInjectProviderClass类注入的特殊处理
            beanName);
             return new
DependencyProviderFactory().createDependencyProvider (descriptor,
        }
        else {
```

```
            //通用处理逻辑
             return doResolveDependency(descriptor,
descriptor.getDependencyType(),
             beanName, autowiredBeanNames, typeConverter);
         }
     }
     protected Object doResolveDependency(DependencyDescriptor
descriptor, Class<?> type,
     String beanName,
     Set<String> autowiredBeanNames, TypeConverter typeConverter)
throws
     BeansException {
         /*
            * 用于支持Spring中新增的注解@Value
         */
          Object value =
getAutowireCandidateResolver().getSuggestedValue(descriptor);
          if (value != null) {
             if (value instanceof String) {
                 String strVal = resolveEmbeddedValue((String) value);
                 BeanDefinition bd = (beanName != null &&
containsBean(beanName) ?
                 getMergedBeanDefinition(beanName) : null);
                 value = evaluateBeanDefinitionString(strVal, bd);
            }
             TypeConverter converter = (typeConverter != null ? typeConverter
:
```

```
getTypeConverter());
 return converter.convertIfNecessary(value, type);
}
//如果解析器没有成功解析，则需要考虑各种情况
//属性是数组类型
 if (type.isArray()) {
  Class<?> componentType = type.getComponentType();
  //根据属性类型找到beanFacotry中所有类型的匹配bean,
  //返回值的构成为：key=匹配的 beanName,value=beanName 对
应的实例化后的 bean(通过
  getBean(beanName)返回)
  Map<String, Object> matchingBeans =
findAutowireCandidates(beanName,
  componentType, descriptor);
  if (matchingBeans.isEmpty()) {
    //如果autowire的require属性为true而找到的匹配项却为空则
只能抛出异常
    if (descriptor.isRequired()) {
      raiseNoSuchBeanDefinitionException(componentType,
"array of " +
      componentType.getName(), descriptor);
    }
    return null;
  }
  if (autowiredBeanNames != null) {
    autowiredBeanNames.addAll(matchingBeans.keySet());
  }
```

```
            TypeConverter converter = (typeConverter != null ? typeConverter
:
            getTypeConverter());
            //通过转换器将bean的值转换为对应的type类型
            return converter.convertIfNecessary(matchingBeans.values(),
type);
        }
        //属性是Collection类型
        else if (Collection.class.isAssignableFrom(type) &&
type.isInterface()) {
            Class<?> elementType = descriptor.getCollectionType();
            if (elementType == null) {
                if (descriptor.isRequired()) {
                    throw new FatalBeanException("No element type declared
for
                    collection [" + type.getName() + "]");
                }
                return null;
            }
                Map<String, Object> matchingBeans =
findAutowireCandidates(beanName,
                elementType, descriptor);
                if (matchingBeans.isEmpty()) {
                    if (descriptor.isRequired()) {
                        raiseNoSuchBeanDefinitionException(elementType,
"collection of "
                        + elementType.getName(), descriptor);
```

```
            }
            return null;
        }
        if (autowiredBeanNames != null) {
           autowiredBeanNames.addAll(matchingBeans.keySet());
        }
        TypeConverter converter = (typeConverter != null ?
typeConverter :
        getTypeConverter());
        return converter.convertIfNecessary(matchingBeans.values(),
type);
    }
    //属性是Map类型
    else if (Map.class.isAssignableFrom(type) && type.isInterface()) {
        Class<?> keyType = descriptor.getMapKeyType();
        if (keyType == null || !String.class.isAssignableFrom(keyType))
{
            if (descriptor.isRequired()) {
                throw new FatalBeanException("Key type [" + keyType +
"] of map
                ["  + type.getName() +
                "] must be assignable to [java.lang.String]");
            }
            return null;
        }
        Class<?> valueType = descriptor.getMapValueType();
        if (valueType == null) {
```

```
            if (descriptor.isRequired()) {
                throw new FatalBeanException("No value type declared
for map ["
                + type.getName() + "]");
            }
            return null;
        }
        Map<String, Object> matchingBeans =
findAutowireCandidates(beanName,
        valueType, descriptor);
        if (matchingBeans.isEmpty()) {
            if (descriptor.isRequired()) {
                raiseNoSuchBeanDefinitionException(valueType, "map
with value
                type " + valueType.getName(), descriptor);
            }
            return null;
        }
        if (autowiredBeanNames != null) {
            autowiredBeanNames.addAll(matchingBeans.keySet());
        }
        return matchingBeans;
    }else {
        Map<String, Object> matchingBeans =
findAutowireCandidates(beanName, type,
        descriptor);
        if (matchingBeans.isEmpty()) {
```

```
                    if (descriptor.isRequired()) {
                        raiseNoSuchBeanDefinitionException(type, "", descriptor);
                    }
                    return null;
                }
                if (matchingBeans.size() > 1) {
                    String primaryBeanName =
determinePrimaryCandidate(matchingBeans,
                    descriptor);
                    if (primaryBeanName == null) {
                        matching bean but found " +
                        throw new NoSuchBeanDefinitionException(type,
"expected single
                        matchingBeans.size() + ": " + matchingBeans.keySet());
                    }
                    if (autowiredBeanNames != null) {
                       autowiredBeanNames.add(primaryBeanName);
                    }
                    return matchingBeans.get(primaryBeanName);
                }
                //已经可以确定只有一个匹配项
                next();
                Map.Entry<String, Object> entry = matchingBeans.entrySet().
iterator().
                if (autowiredBeanNames != null) {
                   autowiredBeanNames.add(entry.getKey());
                }
```

```
            return entry.getValue();
        }
    }
```

寻找类型的匹配执行顺序时，首先尝试使用解析器进行解析，如果解析器没有成功解析，那么可能是使用默认的解析器没有做任何处理，或者是使用了自定义的解析器，但是对于集合等类型来说并不在解析范围之内，所以再次对不同类型进行不同情况的处理，虽说对于不同类型处理方式不一致，但是大致的思路还是很相似的，所以函数中只对数组类型进行了详细地注释。

3．applyPropertyValues

程序运行到这里，已经完成了对所有注入属性的获取，但是获取的属性是以PropertyValues形式存在的，还并没有应用到已经实例化的bean中，这一工作是在applyPropertyValues中。

```
protected void applyPropertyValues(String beanName, BeanDefinition mbd, BeanWrapper bw,
PropertyValues pvs) {
    if (pvs == null || pvs.isEmpty()) {
        return;
    }
    MutablePropertyValues mpvs = null;
    List<PropertyValue> original;
    if (System.getSecurityManager()!= null) {
        if (bw instanceof BeanWrapperImpl) {
            ((BeanWrapperImpl)
bw).setSecurityContext(getAccessControlContext());
        }
    }
```

```
if (pvs instanceof MutablePropertyValues) {
    mpvs = (MutablePropertyValues) pvs;
   //如果mpvs中的值已经被转换为对应的类型那么可以直接设置到beanwapper中
    if (mpvs.isConverted()) {
        // Shortcut: use the pre-converted values as-is.
        try {
            bw.setPropertyValues(mpvs);
            return;
        }
        catch (BeansException ex) {
            throw new BeanCreationException(
            mbd.getResourceDescription(), beanName, "Error setting
            property values", ex);
        }
    }
    original = mpvs.getPropertyValueList();
}else {
   //如果pvs并不是使用MutablePropertyValues封装的类型，那么直接使用原始的属性获取方法
    original = Arrays.asList(pvs.getPropertyValues());
}
TypeConverter converter = getCustomTypeConverter();
if (converter == null) {
    converter = bw;
}
//获取对应的解析器
```

```
BeanDefinitionValueResolver valueResolver = new
BeanDefinitionValueResolver(this,
 beanName, mbd, converter);
 // Create a deep copy, resolving any references for values.
 List<PropertyValue> deepCopy = new ArrayList<PropertyValue>
(original.size());
 boolean resolveNecessary = false;
//遍历属性，将属性转换为对应类的对应属性的类型
 for (PropertyValue pv : original) {
     if (pv.isConverted()) {
        deepCopy.add(pv);
     }else {
         String propertyName = pv.getName();
         Object originalValue = pv.getValue();
         Object resolvedValue =
valueResolver.resolveValueIfNecessary(pv,
        originalValue);
         Object convertedValue = resolvedValue;
         boolean convertible = bw.isWritableProperty(propertyName)
&&
            !PropertyAccessorUtils.isNestedOrIndexedProperty
            (propertyName);
         if (convertible) {
             bw, converter);
             convertedValue = convertForProperty(resolvedValue,
propertyName,
        }
```

```
                if (resolvedValue == originalValue) {
                    if (convertible) {
                        pv.setConvertedValue(convertedValue);
                    }
                    deepCopy.add(pv);
                }
                else if (convertible && originalValue instanceof
TypedStringValue &&
                    !((TypedStringValue) originalValue).isDynamic() &&
                    (convertedValue))) {
                    !(convertedValue instanceof Collection || ObjectUtils.isArray
                    pv.setConvertedValue(convertedValue);
                    deepCopy.add(pv);
                }
                else {
                    resolveNecessary = true;
                    deepCopy.add(new PropertyValue(pv, convertedValue));
                }
            }
        }
        if (mpvs != null && !resolveNecessary) {
            mpvs.setConverted();
        }
        try {
            bw.setPropertyValues(new MutablePropertyValues(deepCopy));
        }
        catch (BeansException ex) {
```

```
throw new BeanCreationException(
values", ex);
mbd.getResourceDescription(), beanName, "Error setting property
}
}
```

5.7.4 初始化bean

大家应该记得在bean配置时bean中有一个init-method的属性，这个属性的作用是在bean实例化前调用init-method指定的方法来根据用户业务进行相应的实例化。我们现在就已经进入这个方法了，首先看一下这个方法的执行位置，Spring中程序已经执行过bean的实例化，并且进行了属性的填充，而就在这时将会调用用户设定的初始化方法。

```
protected Object initializeBean(final String beanName, final Object
bean, RootBean
Definition mbd) {
    if (System.getSecurityManager() != null) {
        AccessController.doPrivileged(new PrivilegedAction<Object>() {
            public Object run() {
                invokeAwareMethods(beanName, bean);
                 return null;
            }
        }, getAccessControlContext());
    }
    else {
        //对特殊的bean处理:Aware、BeanClassLoaderAware、
BeanFactoryAware
        invokeAwareMethods(beanName, bean);
```

```
        }
        Object wrappedBean = bean;
        if (mbd == null || !mbd.isSynthetic()) {
          //应用后处理器
           wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean,
          beanName);
        }
        try {
          //激活用户自定义的init方法
          invokeInitMethods(beanName, wrappedBean, mbd);
        }
        catch (Throwable ex) {
           throw new BeanCreationException(
           (mbd != null ? mbd.getResourceDescription() : null),
           beanName, "Invocation of init method failed", ex);
        }
        if (mbd == null || !mbd.isSynthetic()) {
          //后处理器应用
           wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean,
          beanName);
        }
        return wrappedBean;
    }
```

虽然说此函数的主要目的是进行客户设定的初始化方法的调用，但是除此之外还有些其他必要的工作。

1．激活**Aware**方法

在分析其原理之前，我们先了解一下Aware的使用。Spring中提供一些Aware相关接口，比如BeanFactoryAware、ApplicationContextAware、ResourceLoaderAware、ServletContextAware等，实现这些 Aware 接口的 bean 在被初始之后，可以取得一些相对应的资源，例如实现BeanFactoryAware 的 bean 在初始后， Spring 容器将会注入 BeanFactory 的实例，而实现ApplicationContextAware的bean，在bean被初始后，将会被注入ApplicationContext的实例等。我们首先通过示例方法来了解一下Aware的使用。

（1）定义普通bean。

```
public class Hello {
    public void say() {
        System.out.println("hello");
    }
}
```

（2）定义BeanFactoryAware类型的bean。

```
public class Test implements BeanFactoryAware {
    private BeanFactory beanFactory;
    // 声明bean的时候Spring会自动注入BeanFactory
    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws
BeansException {
        this.beanFactory = beanFactory;
    }
    public void testAware() {
    //通过hello这个bean id从beanFactory获取实例
        Hello hello = (Hello) beanFactory.getBean("hello");
```

```
        hello.say();
    }
}
```

（3）使用main方法测试。

```
public static void main(String[] s) {
    ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
    Test test = (Test) ctx.getBean("test");
    test.testAware();
}
```

运行测试类，控制台输出：

```
hello
```

按照上面的方法我们可以获取到Spring中BeanFactory，并且可以根据BeanFactory获取所有 bean，以及进行相关设置。当然还有其他 Aware的使用方法都大同小异，看一下 Spring的实现方式，相信读者便会使用了。

```
private void invokeAwareMethods(final String beanName, final Object
bean) {
    if (bean instanceof Aware) {
        if (bean instanceof BeanNameAware) {
            ((BeanNameAware) bean).setBeanName(beanName);
        }
        if (bean instanceof BeanClassLoaderAware) {
            ((BeanClassLoaderAware)
bean).setBeanClassLoader(getBeanClassLoader());
        }
        if (bean instanceof BeanFactoryAware) {
```

```
                Factory.this);
                ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowire
CapableBean
            }
        }
    }
```

代码简单得已经没有什么好说的了。读者可以自己尝试使用别的Aware，都比较简单。

2．处理器的应用

BeanPostProcessor相信大家都不陌生，这是Spring中开放式架构中一个必不可少的亮点，给用户充足的权限去更改或者扩展 Spring ，而除了 BeanPostProcessor 外还有很多其他的PostProcessor，当然大部分都是以此为基础，继承自 BeanPostProcessor。BeanPostProcessor 的使用位置就是这里，在调用客户自定义初始化方法前以及调用自定义初始化方法后分别会调用BeanPostProcessor的postProcessBeforeInitialization和postProcessAfterInitialization方法，使用户可以根据自己的业务需求进行响应的处理。

```
    public Object applyBeanPostProcessorsBeforeInitialization(Object
existingBean, String
    beanName)
        throws BeansException {
        Object result = existingBean;
        for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
            result = beanProcessor.postProcessBeforeInitialization(result,
beanName);
            if (result == null) {
                return result;
```

```
        }
      }
      return result;
    }
    public Object applyBeanPostProcessorsAfterInitialization(Object
existingBean, String
    beanName)
      throws BeansException {
      Object result = existingBean;
      for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        result = beanProcessor.postProcessAfterInitialization(result,
beanName);
        if (result == null) {
          return result;
        }
      }
      return result;
    }
```

3．激活自定义的**init**方法

客户定制的初始化方法除了我们熟知的使用配置init-method外，还有使自定义的bean实现InitializingBean接口，并在afterPropertiesSet中实现自己的初始化业务逻辑。

init-method与afterPropertiesSet都是在初始化bean时执行，执行顺序是afterPropertiesSet先执行，而init-method后执行。

在invokeInitMethods方法中就实现了这两个步骤的初始化方法调用。

```
    protected void invokeInitMethods(String beanName, final Object bean,
```

```
RootBeanDefinition mbd)
        throws Throwable {
            //首先会检查是否是InitializingBean，如果是的话需要调用
afterPropertiesSet方法
                boolean isInitializingBean = (bean instanceof InitializingBean);
                if (isInitializingBean && (mbd == null || !mbd.isExternally
ManagedInitMethod
                ("afterPropertiesSet"))) {
                    if (logger.isDebugEnabled()) {
                        logger.debug("Invoking afterPropertiesSet() on bean with
name '" +
                        beanName + "'");
                    }
                    if (System.getSecurityManager() != null) {
                        try {
                            <Object>() {
                            AccessController.doPrivileged(new
PrivilegedExceptionAction
                                public Object run() throws Exception {
                                    ((InitializingBean) bean).afterPropertiesSet();
                                    return null;
                                }
                            }, getAccessControlContext());
                        }
                        catch (PrivilegedActionException pae) {
                            throw pae.getException();
                        }
```

```
            }else {
              //属性初始化后的处理
               ((InitializingBean) bean).afterPropertiesSet();
            }
          }
          if (mbd != null) {
              String initMethodName = mbd.getInitMethodName();
              if (initMethodName != null && !(isInitializingBean &&
"afterPropertiesSet".
          equals(initMethodName)) &&
                 !mbd.isExternallyManagedInitMethod(initMethodName)) {
            //调用自定义初始化方法
              invokeCustomInitMethod(beanName, bean, mbd);
          }
        }
    }
```

5.7.5 注册DisposableBean

Spring中不但提供了对于初始化方法的扩展入口，同样也提供了销毁方法的扩展入口，对于销毁方法的扩展，除了我们熟知的配置属性destroy-method方法外，用户还可以注册后处理器DestructionAwareBeanPostProcessor来统一处理bean的销毁方法，代码如下：

```
protected void registerDisposableBeanIfNecessary(String beanName,
Object bean,
RootBeanDefinition mbd) {
    AccessControlContext acc = (System.getSecurityManager() != null ?
```

```
getAccess
            ControlContext() : null);
            if (!mbd.isPrototype() && requiresDestruction(bean, mbd)) {
                if (mbd.isSingleton()) {
                    /*
                     * 单例模式下注册需要销毁的bean，此方法中会处理实现
DisposableBean的bean,
                     * 并且对所有的bean使用
DestructionAwareBeanPostProcessors处理
                     * DisposableBean DestructionAwareBeanPostProcessors,
                     */
                    registerDisposableBean(beanName,
                     new DisposableBeanAdapter(bean, beanName, mbd,
getBeanPost
                     Processors(), acc));
                }else {
                    /*
                     * 自定义scope的处理
                     */
                     Scope scope = this.scopes.get(mbd.getScope());
                     if (scope == null) {
                         throw new IllegalStateException("No Scope registered for
scope '"
                         + mbd.getScope() + "'");
                    }
                    scope.registerDestructionCallback(beanName,
                     new DisposableBeanAdapter(bean, beanName, mbd,
```

```
getBeanPost
                    Processors(), acc));
                }
            }
        }
```

第6章　容器的功能扩展

经过前面几章的分析，相信大家已经对Spring中的容器功能有了简单的了解，在前面的章节中我们一直以BeanFacotry接口以及它的默认实现类XmlBeanFactory为例进行分析，但是， Spring中还提供了另一个接口ApplicationContext，用于扩展BeanFacotry中现有的功能。

ApplicationContext和BeanFacotry两者都是用于加载Bean的，但是相比之下，Application Context提供了更多的扩展功能，简单一点说：ApplicationContext包含BeanFactory的所有功能。通常建议比BeanFactory 优先，除非在一些限制的场合，比如字节长度对内存有很大的影响时（Applet）。绝大多数“典型的”企业应用和系统，ApplicationContext就是你需要使用的。

那么究竟 ApplicationContext 比 BeanFactory 多出了哪些功能呢？还需要我们进一步的探索。首先我们来看看使用两个不同的类去加载配置文件在写法上的不同。

使用BeanFactory方式加载XML。

BeanFactory bf = new XmlBeanFactory(new ClassPathResource("beanFactoryTest.xml"));

使用ApplicationContext方式加载XML。

ApplicationContext bf = new ClassPathXmlApplicationContext("beanFactoryTest.xml");

同样，我们还是以ClassPathXmlApplicationContext作为切入点，开始对整体功能进行分析。

```
public ClassPathXmlApplicationContext(String configLocation) throws
BeansException {
    this(new String[] {configLocation}, true, null);
}
public ClassPathXmlApplicationContext(String[] configLocations,
boolean refresh,
ApplicationContext parent) throws BeansException {
    super(parent);
    setConfigLocations(configLocations);
     if (refresh) {
        refresh();
    }
}
```

设置路径是必不可少的步骤，ClassPathXmlApplicationContext中可以将配置文件路径以数组的方式传入，ClassPathXmlApplicationContext可以对数组进行解析并进行加载。而对于解析及功能实现都在refresh()中实现。

6.1 设置配置路径

在ClassPathXmlApplicationContext中支持多个配置文件以数组方式同时传入：

```
public void setConfigLocations(String[] locations) {
     if (locations != null) {
        Assert.noNullElements(locations, "Config locations must not be
```

```
null");
            this.configLocations = new String[locations.length];
            for (int i = 0; i < locations.length; i++) {
           //解析给定路径
               this.configLocations[i] = resolvePath(locations[i]).trim();
           }
        }
        else {
            this.configLocations = null;
        }
    }
```

此函数主要用于解析给定的路径数组，当然，如果数组中包含特殊符号，如${var}，那么在resolvePath中会搜寻匹配的系统变量并替换。

6.2 扩展功能

设置了路径之后，便可以根据路径做配置文件的解析以及各种功能的实现了。可以说refresh 函数中包含了几乎 ApplicationContext 中提供的全部功能，而且此函数中逻辑非常清晰明了，使我们很容易分析对应的层次及逻辑。

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
      //准备刷新的上下文环境
      prepareRefresh();
       // Tell the subclass to refresh the internal bean factory.
      //初始化BeanFactory，并进行XML文件读取
       ConfigurableListableBeanFactory beanFactory =
```

obtainFreshBeanFactory();

// Prepare the bean factory for use in this context.

//对BeanFactory进行各种功能填充

prepareBeanFactory(beanFactory);

try {

// Allows post-processing of the bean factory in context subclasses.

// 子类覆盖方法做额外的处理

postProcessBeanFactory(beanFactory);

//激活各种BeanFactory处理器

invokeBeanFactoryPostProcessors(beanFactory);

// 注册拦截Bean创建的Bean处理器,这里只是注册，真正的调用是在getBean时候

registerBeanPostProcessors(beanFactory);

// 为上下文初始化Message源，即不同语言的消息体 ，国际化处理

initMessageSource();

// Initialize event multicaster for this context.

// 初始化应用消息广播器，并放入“applicationEventMulticaster”bean中

initApplicationEventMulticaster();

// Initialize other special beans in specific context subclasses.

// 留给子类来初始化其它的Bean

onRefresh();

// Check for listener beans and register them.

//在所有注册的bean中查找Listener bean，注册到消息广播器中

```
                registerListeners();
                 // Instantiate all remaining (non-lazy-init) singletons.
                // 初始化剩下的单实例（非惰性的）
                finishBeanFactoryInitialization(beanFactory);
                 // Last step: publish corresponding event.
                // 完成刷新过程，通知生命周期处理器 lifecycleProcessor 刷
新过程，同时发出
                ContextRefreshEvent通知别人
                finishRefresh();
            }
            catch (BeansException ex) {
                 // Destroy already created singletons to avoid dangling
resources.
                destroyBeans();
                 // Reset 'active' flag.
                cancelRefresh(ex);
                 // Propagate exception to caller.
                 throw ex;
            }
        }
    }
```

下面概括一下ClassPathXmlApplicationContext初始化的步骤，并从中解释一下它为我们提供的功能。

（1）初始化前的准备工作，例如对系统属性或者环境变量进行准备及验证。

在某种情况下项目的使用需要读取某些系统变量，而这个变量的设置很可能会影响着系统的正确性，那么ClassPathXmlApplicationContext

为我们提供的这个准备函数就显得非常必要，它可以在Spring启动的时候提前对必须的变量进行存在性验证。

（2）初始化BeanFactory，并进行XML文件读取。

之前有提到ClassPathXmlApplicationContext包含着BeanFactory所提供的一切特征，那么在这一步骤中将会复用 BeanFactory 中的配置文件读取解析及其他功能，这一步之后， ClassPathXmlApplicationContext 实际上就已经包含了 BeanFactory 所提供的功能，也就是可以进行Bean的提取等基础操作了。

（3）对BeanFactory进行各种功能填充。

@Qualifier与@Autowired应该是大家非常熟悉的注解，那么这两个注解正是在这一步骤中增加的支持。

（4）子类覆盖方法做额外的处理。

Spring之所以强大，为世人所推崇，除了它功能上为大家提供了便例外，还有一方面是它的完美架构，开放式的架构让使用它的程序员很容易根据业务需要扩展已经存在的功能。这种开放式的设计在 Spring 中随处可见，例如在本例中就提供了一个空的函数实现 postProcess BeanFactory来方便程序员在业务上做进一步扩展。

（5）激活各种BeanFactory处理器。

（6）注册拦截bean创建的bean处理器，这里只是注册，真正的调用是在getBean时候。

（7）为上下文初始化Message源，即对不同语言的消息体进行国际化处理。

（8）初始化应用消息广播器，并放入“applicationEventMulticaster”bean中。

（9）留给子类来初始化其他的bean。

（10）在所有注册的bean中查找 listener bean，注册到消息广播器中。

（11）初始化剩下的单实例（非惰性的）。

（12）完成刷新过程，通知生命周期处理器lifecycleProcessor刷新过程，同时发出Context RefreshEvent通知别人。

6.3 环境准备

prepareRefresh函数主要是做些准备工作，例如对系统属性及环境变量的初始化及验证。

```
protected void prepareRefresh() {
    this.startupDate = System.currentTimeMillis();
    synchronized (this.activeMonitor) {
        this.active = true;
    }
    if (logger.isInfoEnabled()) {
        logger.info("Refreshing " + this);
    }
    //留给子类覆盖
    initPropertySources();
    //验证需要的属性文件是否都已经放入环境中
    getEnvironment().validateRequiredProperties();
}
```

网上有人说其实这个函数没什么用，因为最后两句代码才是最为关键的，但是却没有什么逻辑处理，initPropertySources是空的，没有任何逻辑，而getEnvironment().validateRequiredProperties也因为没有需要验证的属性而没有做任何处理。其实这都是因为没有彻底理解才会这么说，这个函数如果用好了作用还是挺大的。那么，该怎么用呢？我们先探索下各个函数的作用。

（1）initPropertySources正符合Spring的开放式结构设计，给用户最大扩展Spring的能力。用户可以根据自身的需要重写 initPropertySources 方法，并在方法中进行个性化的属性处理及设置。

（2）validateRequiredProperties 则是对属性进行验证，那么如何验证呢？我们举个融合两句代码的小例子来帮助大家理解。

假如现在有这样一个需求，工程在运行过程中用到的某个设置（例如VAR）是从系统环境变量中取得的，而如果用户没有在系统环境变量中配置这个参数，那么工程可能不会工作。这一要求可能会有各种各样的解决办法，当然，在Spring中可以这样做，你可以直接修改Spring的源码，例如修改ClassPathXmlApplicationContext。当然，最好的办法还是对源码进行扩展，我们可以自定义类：

```
public class MyClassPathXmlApplicationContext extends
ClassPathXmlApplicationContext{
     public MyClassPathXmlApplicationContext(String...
configLocations ){
        super(configLocations);
     }
     protected void initPropertySources() {
    //添加验证要求
        getEnvironment().setRequiredProperties("VAR");
     }
}
```

我们自定义了继承自ClassPathXmlApplicationContext的MyClassPathXmlApplicationContext，并重写了 initPropertySources 方法，在方法中添加了我们的个性化需求，那么在验证的时候也就是程序走到getEnvironment().validateRequiredProperties()代码的时候，如果系统并没有检测到对应 VAR 的环境变量，那么将抛出异常。当然我们还需

要在使用的时候替换掉原有的ClassPathXmlApplicationContext：

```
public static void main(String[] args) {
    ApplicationContext bf = new MyClassPathXmlApplicationContext
("test/customtag/
    test.xml");
    User user=(User) bf.getBean("testbean");
}
```

6.4 加载BeanFactory

obtainFreshBeanFactory 方法从字面理解是获取 BeanFactory。之前有说过， Application Context是对BeanFactory的功能上的扩展，不但包含了BeanFactory的全部功能更在其基础上添加了大量的扩展应用，那么obtainFreshBeanFactory正是实现BeanFactory的地方，也就是经过了这个函数后ApplicationContext就已经拥有了BeanFactory的全部功能。

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    //初始化BeanFactory，并进行XML文件读取,并将得到的
BeanFacotry记录在当前实体的属性中
    refreshBeanFactory();
//返回当前实体的beanFactory属性
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (logger.isDebugEnabled()) {
        logger.debug("Bean factory for " + getDisplayName() + ": " +
beanFactory);
    }
    return beanFactory;
}
```

方法中将核心实现委托给了refreshBeanFactory：

```
@Override
protected final void refreshBeanFactory() throws BeansException {
    if (hasBeanFactory()) {
        destroyBeans();
        closeBeanFactory();
    }
    try {
        //创建DefaultListableBeanFactory
         DefaultListableBeanFactory beanFactory = createBeanFactory();
        //为了序列化指定id，如果需要的话,让这个BeanFactory从id反
序列化到BeanFactory对象
        beanFactory.setSerializationId(getId());
        //定制beanFactory，设置相关属性，包括是否允许覆盖同名称
的不同定义的对象以及循环依赖以及
         //设置 @Autowired 和 @Qualifier 注解解析器
QualifierAnnotationAutowire
        CandidateResolver
        customizeBeanFactory(beanFactory);
        //初始化DodumentReader，并进行XML文件读取及解析
        loadBeanDefinitions(beanFactory);
         synchronized (this.beanFactoryMonitor) {
            this.beanFactory = beanFactory;
        }
    }
    catch (IOException ex) {
         throw new ApplicationContextException("I/O error parsing bean
```

definition

source for " + getDisplayName(), ex);

}

}

我们详细分析上面的每个步骤。

（1）创建DefaultListableBeanFactory。

在介绍BeanFactory的时候，不知道读者是否还有印象，声明方式为：BeanFactory bf = new XmlBeanFactory("beanFactoryTest.xml")，其中的XmlBeanFactory继承自DefaultListableBean Factory，并提供了XmlBeanDefinitionReader类型的reader属性，也就是说DefaultListableBean Factory是容器的基础。必须首先要实例化，那么在这里就是实例化DefaultListableBeanFactory的步骤。

（2）指定序列化ID。

（3）定制BeanFactory。

（4）加载BeanDefinition。

（5）使用全局变量记录BeanFactory类实例。

因为DefaultListableBeanFactory类型的变量beanFactory是函数内的局部变量，所以要使用全局变量记录解析结果。

6.4.1 定制BeanFactory

这里已经开始了对 BeanFactory 的扩展，在基本容器的基础上，增加了是否允许覆盖是否允许扩展的设置并提供了注解@Qualifier和@Autowired的支持。

protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {

//如果属性allowBeanDefinitionOverriding不为空，设置给beanFactory对象相应属性，

//此属性的含义：是否允许覆盖同名称的不同定义的对象
 if (this.allowBeanDefinitionOverriding != null) {

beanFactory.setAllowBeanDefinitionOverriding(this.allowBeanDefinitionOve
}
//如果属性allowCircularReferences不为空，设置给beanFactory对象相应属性，
//此属性的含义：是否允许bean之间存在循环依赖
 if (this.allowCircularReferences != null) {

beanFactory.setAllowCircularReferences(this.allowCircularReferences);
}
//用于@Qualifier和@Autowired
beanFactory.setAutowireCandidateResolver(new QualifierAnnotationAutowire
 Candidate Resolver());
}

对于允许覆盖和允许依赖的设置这里只是判断了是否为空，如果不为空要进行设置，但是并没有看到在哪里进行设置，究竟这个设置是在哪里进行设置的呢？还是那句话，使用子类覆盖方法，例如：

public class MyClassPathXmlApplicationContext extends ClassPathXmlApplicationContext{
 ... ...
 protected void customizeBeanFactory(DefaultListableBeanFactory beanFactory) {
  super.setAllowBeanDefinitionOverriding(false);
  super.setAllowCircularReferences(false);

```
        super.customizeBeanFactory(beanFactory);
    }
}
```

设置完后相信大家已经对于这两个属性的使用有所了解，或者可以回到前面的章节进行再一次查看。对于定制 BeanFactory，Spring 还提供了另外一个重要的扩展，就是设置 Autowire CandidateResolver，在bean加载部分中讲解创建Bean时，如果采用autowireByType方式注入，那么默认会使用Spring提供的SimpleAutowireCandidateResolver，而对于默认的实现并没有过多的逻辑处理。在这里，Spring使用了QualifierAnnotationAutowireCandidateResolver，设置了这个解析器后Spring就可以支持注解方式的注入了。

在讲解根据类型自定注入的时候，我们说过解析autowire类型时首先会调用方法：

```
Object value =
getAutowireCandidateResolver().getSuggestedValue(descriptor);
```

因此我们知道，在 QualifierAnnotationAutowireCandidateResolver 中一定会提供了解析Qualifier与Autowire注解的方法。

QualifierAnnotationAutowireCandidateResolver.java

```
public Object getSuggestedValue(DependencyDescriptor descriptor) {
    Object value = findValue(descriptor.getAnnotations());
    if (value == null) {
        MethodParameter methodParam =
descriptor.getMethodParameter();
        if (methodParam != null) {
            value = findValue(methodParam.getMethodAnnotations());
        }
    }
```

```
        return value;
    }
```

6.4.2 加载BeanDefinition

在第一步中提到了将ClassPathXmlApplicationContext与XmlBeanFactory创建的对比，在实现配置文件的加载功能中除了我们在第一步中已经初始化的DefaultListableBeanFactory外，还需要XmlBeanDefinitionReader 来读取 XML ，那么在这个步骤中首先要做的就是初始化XmlBeanDefinitionReader。

```
@Override
    protected void loadBeanDefinitions(DefaultListableBeanFactory beanFactory) throws
        BeansException, IOException {
        //为指定beanFactory创建XmlBeanDefinitionReader
         XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
        //对beanDefinitionReader进行环境变量的设置
        beanDefinitionReader.setEnvironment(this.getEnvironment());
        beanDefinitionReader.setResourceLoader(this);
         beanDefinitionReader.setEntityResolver(new ResourceEntityResolver(this));
        //对BeanDefinitionReader进行设置，可以覆盖
        initBeanDefinitionReader(beanDefinitionReader);
        loadBeanDefinitions(beanDefinitionReader);
    }
```

在初始化了DefaultListableBeanFactory和XmlBeanDefinitionReader后就可以进行配置文件的读取了。

```
protected void loadBeanDefinitions(XmlBeanDefinitionReader reader)
throws BeansException,
    IOException {
        Resource[] configResources = getConfigResources();
        if (configResources != null) {
            reader.loadBeanDefinitions(configResources);
        }
        String[] configLocations = getConfigLocations();
        if (configLocations != null) {
            reader.loadBeanDefinitions(configLocations);
        }
    }
```

使用XmlBeanDefinitionReader的loadBeanDefinitions方法进行配置文件的加载机注册相信大家已经不陌生，这完全就是开始BeanFactory的套路。因为在XmlBeanDefinitionReader中已经将之前初始化的DefaultListableBeanFactory注册进去了，所以XmlBeanDefinitionReader所读取的BeanDefinitionHolder都会注册到DefaultListableBeanFactory中，也就是经过此步骤，类型DefaultListableBeanFactory的变量beanFactory已经包含了所有解析好的配置。

6.5 功能扩展

进入函数prepareBeanFactory前，Spring已经完成了对配置的解析，而ApplicationContext在功能上的扩展也由此展开。

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory
beanFactory) {
        //设置beanFactory的classLoader为当前context的classLoader
```

beanFactory.setBeanClassLoader(getClassLoader());
//设置beanFactory的表达式语言处理器，Spring3增加了表达式语言的支持，
//默认可以使用#{bean.xxx}的形式来调用相关属性值。
beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver());
//为beanFactory增加了一个默认的propertyEditor，这个主要是对bean的属性等设置管理的一
个工具
beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this,
getEnvironment()));
/*
* 添加BeanPostProcessor,
*/
beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
//设置了几个忽略自动装配的接口

beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);

beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.cla

beanFactory.ignoreDependencyInterface(MessageSourceAware.class);

beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
beanFactory.ignoreDependencyInterface(EnvironmentAware.class);

```
		//设置了几个自动装配的特殊规则
		 beanFactory.registerResolvableDependency(BeanFactory.class,
beanFactory);
		 beanFactory.registerResolvableDependency(ResourceLoader.class,
this);

beanFactory.registerResolvableDependency(ApplicationEventPublisher.class,
this);

beanFactory.registerResolvableDependency(ApplicationContext.class, this);
		//增加对AspectJ的支持
		 if
(beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
			beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor(beanFactory));
			// Set a temporary ClassLoader for type matching.
			beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader (beanFactory.
		   getBeanClassLoader()));
		}
		//添加默认的系统环境bean
		 if
(!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
			beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME,
getEnvironment());
		}
		 if
```

```
(!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {

beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().
        getSystemProperties());
      }
      if
(!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAM
{

beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME,
getEnvironment().
        getSystemEnvironment());
      }
    }
```

上面函数中主要进行了几个方面的扩展。

增加对SPEL语言的支持。

增加对属性编辑器的支持。

增加对一些内置类，比如EnvironmentAware、MessageSourceAware的信息注入。

设置了依赖功能可忽略的接口。

注册一些固定依赖的属性。

增加AspectJ的支持（会在第7章中进行详细的讲解）。

将相关环境变量及属性注册以单例模式注册。

可能读者不是很理解每个步骤的具体含义，接下来我们会对各个步骤进行详细地分析。

6.5.1 增加SPEL语言的支持

Spring表达式语言全称为“Spring Expression Language”，缩写为“SpEL”，类似于Struts 2x中使用的OGNL表达式语言，能在运行时构建复杂表达式、存取对象图属性、对象方法调用等，并且能与Spring功能完美整合，比如能用来配置bean定义。SpEL是单独模块，只依赖于core模块，不依赖于其他模块，可以单独使用。

SpEL使用#{…}作为定界符，所有在大框号中的字符都将被认为是SpEL，使用格式如下：

```
<bean id="saxophone" value="com.xxx.xxx.Xxx"/>
<bean >
    <property name="instrument" value="#{saxophone}"/>
<bean/>
```

相当于：

```
<bean id="saxophone" value="com.xxx.xxx.Xxx"/>
<bean >
    <property name="instrument" ref="saxophone"/>
<bean/>
```

当然，上面只是列举了其中最简单的使用方式，SPEL功能非常强大，使用好可以大大提高开发效率，这里只为唤起读者的记忆来帮助我们理解源码，有兴趣的读者可以进一步深入研究。

在源码中通过代码 beanFactory.setBeanExpressionResolver(new StandardBeanExpression Resolver())注册语言解析器，就可以对SPEL进行解析了，那么在注册解析器后Spring又是在什么时候调用这个解析器进行解析呢？

之前我们讲解过 Spring 在 bean 进行初始化的时候会有属性填充的一步，而在这一步中Spring会调用AbstractAutowireCapableBeanFactory

类的applyPropertyValues函数来完成功能。就在这个函数中，会通过构造BeanDefinitionValueResolver类型实例valueResolver来进行属性值的解析。同时，也是在这个步骤中一般通过 AbstractBeanFactory 中的evaluateBeanDefinitionString方法去完成SPEL的解析。

```
protected Object evaluateBeanDefinitionString(String value,
BeanDefinition beanDefinition) {
    if (this.beanExpressionResolver == null) {
        return value;
    }
    Scope scope = (beanDefinition != null ?
getRegisteredScope(beanDefinition.getScope()) :
    null);
    return this.beanExpressionResolver.evaluate(value, new
BeanExpressionContext(this,
    scope));
}
```

当调用这个方法时会判断是否存在语言解析器，如果存在则调用语言解析器的方法进行解析，解析的过程是在Spring的expression的包内，这里不做过多解释。我们通过查看对evaluate BeanDefinitionString方法的调用层次可以看出，应用语言解析器的调用主要是在解析依赖注入bean的时候，以及在完成bean的初始化和属性获取后进行属性填充的时候。

6.5.2 增加属性注册编辑器

在Spring DI注入的时候可以把普通属性注入进来，但是像Date类型就无法被识别。例如：

```
public class UserManager {
    private Date dataValue;
```

```
    public Date getDataValue() {
        return dataValue;
    }
public void setDataValue(Date dataValue) {
        this.dataValue = dataValue;
    }
public String toString(){
        return "dataValue: " + dataValue;
    }
}
```

上面代码中，需要对日期型属性进行注入：

```
<bean id="userManager" class="com.test.UserManager">
    <property name="dataValue">
      <value>2013-03-15</value>
   </property>
</bean>
```

测试代码：

```
@Test
public void testDate(){
    ApplicationContext ctx = new
ClassPathXmlApplicationContext("beans.xml");
    UserManager userManager =
(UserManager)ctx.getBean("userManager");
   System.out.println(userManager);
}
```

如果直接这样使用，程序则会报异常，类型转换不成功。因为在UserManager 中的dataValue属性是Date类型的，而在XML中配置的却是

String类型的，所以当然会报异常。

Spring针对此问题提供了两种解决办法。

1．使用自定义属性编辑器

使用自定义属性编辑器，通过继承PropertyEditorSupport，重写setAsText方法，具体步骤如下。

（1）编写自定义的属性编辑器。

```
public class DatePropertyEditor extends PropertyEditorSupport {
    private String format = "yyyy-MM-dd";
    public void setFormat(String format) {
        this.format = format;
    }
    public void setAsText(String arg0) throws IllegalArgumentException
{
        System.out.println("arg0: " + arg0);
        SimpleDateFormat sdf = new SimpleDateFormat(format);
        try {
            Date d = sdf.parse(arg0);
            this.setValue(d);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}
```

（2）将自定义属性编辑器注册到Spring中。

```
<!--自定义属性编辑器 -->
<bean
class="org.Springframework.beans.factory.config.CustomEditorConfigurer">
```

```
    <property name="customEditors">
      <map>
       <entry key="java.util.Date">
       <bean class="com.test.DatePropertyEditor">
           <property name="format" value="yyyy-MM-dd"/>
      </bean>
  </entry>
      </map>
  </property>
</bean>
```

在配置文件中引入类型为org.Springframework.beans.factory.config.CustomEditorConfigurer的bean，并在属性customEditors中加入自定义的属性编辑器，其中key为属性编辑器所对应的类型。通过这样的配置，当Spring在注入bean的属性时一旦遇到了java.util.Date类型的属性会自动调用自定义的DatePropertyEditor 解析器进行解析，并用解析结果代替配置属性进行注入。

2．注册**Spring**自带的属性编辑器**CustomDateEditor**

通过注册Spring自带的属性编辑器CustomDateEditor，具体步骤如下。

（1）定义属性编辑器。

```
public class DatePropertyEditorRegistrar implements PropertyEditorRegistrar{
    public void registerCustomEditors(PropertyEditorRegistry registry) {
       registry.registerCustomEditor(Date.class, new CustomDateEditor(new SimpleDateFormat
      ("yyyy-MM-dd"),true));
```

```
    }
}
```

（2）注册到Spring中。

```
<!--注册Spring自带编辑器 -->
<bean
class="org.Springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <bean class="com.test.DatePropertyEditorRegistrar"></bean>
        </list>
    </property>
</bean>
```

通过在配置文件中将自定义的 DatePropertyEditorRegistrar 注册进入 org.Springframework. beans.factory.config.CustomEditorConfigurer的propertyEditorRegistrars属性中，可以具有与方法1同样的效果。

我们了解了自定义属性编辑器的使用，但是，似乎这与本节中围绕的核心代码beanFactory.add PropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()))并无联系，因为在注册自定义属性编辑器的时候使用的是PropertyEditorRegistry的registerCustomEditor方法，而这里使用的是ConfigurableListableBeanFactory的addPropertyEditorRegistrar方法。我们不妨深入探索一下ResourceEditorRegistrar的内部实现，在ResourceEditorRegistrar中，我们最关心的方法是registerCustomEditors。

```
public void registerCustomEditors(PropertyEditorRegistry registry) {
    ResourceEditor baseEditor = new
ResourceEditor(this.resourceLoader, this.
    propertyResolver);
```

```
            doRegisterEditor(registry, Resource.class, baseEditor);
            doRegisterEditor(registry, ContextResource.class, baseEditor);
            doRegisterEditor(registry, InputStream.class, new
InputStreamEditor(baseEditor));
            doRegisterEditor(registry, InputSource.class, new
InputSourceEditor(baseEditor));
            doRegisterEditor(registry, File.class, new FileEditor(baseEditor));
            doRegisterEditor(registry, URL.class, new URLEditor(baseEditor));
            ClassLoader classLoader = this.resourceLoader.getClassLoader();
            doRegisterEditor(registry, URI.class, new URIEditor(classLoader));
            doRegisterEditor(registry, Class.class, new
ClassEditor(classLoader));
            doRegisterEditor(registry, Class[].class, new
ClassArrayEditor(classLoader));
            if (this.resourceLoader instanceof ResourcePatternResolver) {
                doRegisterEditor(registry, Resource[].class,
                new ResourceArrayPropertyEditor((ResourcePatternResolver)
this.
                resourceLoader, this.propertyResolver));
            }
        }
        private void doRegisterEditor(PropertyEditorRegistry registry, Class<?>
requiredType,
        PropertyEditor editor) {
            if (registry instanceof PropertyEditorRegistrySupport) {
                ((PropertyEditorRegistrySupport) registry).overrideDefaultEditor
(requiredType,
```

```
            editor);
        }
        else {
            registry.registerCustomEditor(requiredType, editor);
        }
    }
```

在 doRegisterEditor 函数中，可以看到在之前提到的自定义属性中使用的关键代码：registry.registerCustomEditor(requiredType, editor)，回过头来看 ResourceEditorRegistrar 类的registerCustomEditors方法的核心功能，其实无非是注册了一系列的常用类型的属性编辑器，例如，代码doRegisterEditor(registry, Class.class, new ClassEditor(classLoader))实现的功能就是注册Class类对应的属性编辑器。那么，注册后，一旦某个实体bean中存在一些Class类型的属性，那么Spring会调用ClassEditor将配置中定义的String类型转换为Class类型并进行赋值。

分析到这里，我们不禁有个疑问，虽说ResourceEditorRegistrar类的registerCustomEditors方法实现了批量注册的功能，但是beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()))仅仅是注册了ResourceEditorRegistrar实例，却并没有调用ResourceEditorRegistrar的 registerCustomEditors 方法进行注册，那么到底是在什么时候进行注册的呢？进一步查看ResourceEditorRegistrar的registerCustomEditors方法的调用层次结构，如图6-1所示。



图6-1 ResourceEditorRegistrar的 registerCustomEditors方法的调用层次结构

发现在 AbstractBeanFactory 中的 registerCustomEditors 方法中被调

用过，继续查看AbstractBeanFactory中的registerCustomEditors方法的调用层次结构，如图6-2所示。



图6-2 AbstractBeanFactory中的 registerCustomEditors方法的调用层次结构

其中我们看到一个方法是我们熟悉的，就是 AbstractBeanFactory 类中的 initBeanWrapper方法，这是在 bean 初始化时使用的一个方法，之前已经使用过大量的篇幅进行讲解，主要是在将BeanDefinition转换为BeanWrapper后用于对属性的填充。到此，逻辑已经明了，在bean的初始化后会调用ResourceEditorRegistrar的registerCustomEditors方法进行批量的通用属性编辑器注册。注册后，在属性填充的环节便可以直接让Spring使用这些编辑器进行属性的解析了。

既然提到了BeanWrapper，这里也有必要强调下，Spring中用于封装bean的是BeanWrapper类型，而它又间接继承了 PropertyEditorRegistry类型，也就是我们之前反复看到的方法参数PropertyEditorRegistry registry，其实大部分情况下都是 BeanWrapper，对于 BeanWrapper 在Spring中的默认实现是BeanWrapperImpl，而BeanWrapperImpl除了实现BeanWrapper接口外还继承了PropertyEditorRegistrySupport，在PropertyEditorRegistrySupport中有这样一个方法：

```
private void createDefaultEditors() {
    this.defaultEditors = new HashMap<Class<?>, PropertyEditor>(64);
    // Simple editors, without parameterization capabilities.
    // The JDK does not contain a default editor for any of these target
types.
```

this.defaultEditors.put(Charset.class, new CharsetEditor());
this.defaultEditors.put(Class.class, new ClassEditor());
this.defaultEditors.put(Class[].class, new ClassArrayEditor());
this.defaultEditors.put(Currency.class, new CurrencyEditor());
this.defaultEditors.put(File.class, new FileEditor());
this.defaultEditors.put(InputStream.class, new InputStreamEditor());
this.defaultEditors.put(InputSource.class, new InputSourceEditor());
this.defaultEditors.put(Locale.class, new LocaleEditor());
this.defaultEditors.put(Pattern.class, new PatternEditor());
this.defaultEditors.put(Properties.class, new PropertiesEditor());
this.defaultEditors.put(Resource[].class, new
ResourceArrayPropertyEditor());
this.defaultEditors.put(TimeZone.class, new TimeZoneEditor());
this.defaultEditors.put(URI.class, new URIEditor());
this.defaultEditors.put(URL.class, new URLEditor());
this.defaultEditors.put(UUID.class, new UUIDEditor());
// Default instances of collection editors.
// Can be overridden by registering custom instances of those as
custom editors.
class));
this.defaultEditors.put(Collection.class, new CustomCollectionEditor
(Collection.
this.defaultEditors.put(Set.class, new
CustomCollectionEditor(Set.class));
class));
this.defaultEditors.put(SortedSet.class, new CustomCollectionEditor
(SortedSet.

this.defaultEditors.put(List.class, new
CustomCollectionEditor(List.class));
this.defaultEditors.put(SortedMap.class, new
CustomMapEditor(SortedMap.class));
// Default editors for primitive arrays.
this.defaultEditors.put(byte[].class, new ByteArrayPropertyEditor());
this.defaultEditors.put(char[].class, new CharArrayPropertyEditor());
// The JDK does not contain a default editor for char!
this.defaultEditors.put(char.class, new CharacterEditor(false));
this.defaultEditors.put(Character.class, new CharacterEditor(true));
// Spring's CustomBooleanEditor accepts more flag values than the
JDK's default editor.
this.defaultEditors.put(boolean.class, new
CustomBooleanEditor(false));
this.defaultEditors.put(Boolean.class, new
CustomBooleanEditor(true));
// The JDK does not contain default editors for number wrapper
types!
// Override JDK primitive number editors with our own
CustomNumberEditor.
this.defaultEditors.put(byte.class, new
CustomNumberEditor(Byte.class, false));
this.defaultEditors.put(Byte.class, new
CustomNumberEditor(Byte.class, true));
this.defaultEditors.put(short.class, new
CustomNumberEditor(Short.class, false));
this.defaultEditors.put(Short.class, new

```
CustomNumberEditor(Short.class, true));
        this.defaultEditors.put(int.class, new
CustomNumberEditor(Integer.class, false));
        this.defaultEditors.put(Integer.class, new
CustomNumberEditor(Integer.class, true));
        this.defaultEditors.put(long.class, new
CustomNumberEditor(Long.class, false));
        this.defaultEditors.put(Long.class, new
CustomNumberEditor(Long.class, true));
        this.defaultEditors.put(float.class, new
CustomNumberEditor(Float.class, false));
        this.defaultEditors.put(Float.class, new
CustomNumberEditor(Float.class, true));
        this.defaultEditors.put(double.class, new
CustomNumberEditor(Double.class, false));
        this.defaultEditors.put(Double.class, new
CustomNumberEditor(Double.class, true));
        this.defaultEditors.put(BigDecimal.class, new
CustomNumberEditor(BigDecimal.
        class, true));
        this.defaultEditors.put(BigInteger.class, new
CustomNumberEditor(BigInteger.
        class, true));
        // Only register config value editors if explicitly requested.
        if (this.configValueEditorsActive) {
          StringArrayPropertyEditor sae = new StringArrayPropertyEditor();
          this.defaultEditors.put(String[].class, sae);
```

```
            this.defaultEditors.put(short[].class, sae);
            this.defaultEditors.put(int[].class, sae);
            this.defaultEditors.put(long[].class, sae);
        }
    }
```

具体的调用方法我们就不去深究了，但是至少通过这个方法我们已经知道了在 Spring 中定义了上面一系列常用的属性编辑器使我们可以方便地进行配置。如果我们定义的 bean 中的某个属性的类型不在上面的常用配置中的话，才需要我们进行个性化属性编辑器的注册。

6.5.3 添加ApplicationContextAwareProcessor处理器

了解了属性编辑器的使用后，接下来我们继续通过 AbstractApplicationContext 的prepareBeanFactory 方法的主线来进行函数跟踪。对于 beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this))其实主要目的就是注册个 BneaPostProcessor，而真正的逻辑还是在 ApplicationContextAwareProcessor中。

ApplicationContextAwareProcessor实现BeanPostProcessor接口，我们回顾下之前讲过的内容，在bean实例化的时候，也就是Spring激活bean的init-method的前后，会调用BeanPost Processor的postProcessBeforeInitialization方法和postProcessAfterInitialization方法。同样，对于ApplicationContextAwareProcessor我们也关心这两个方法。

对于postProcessAfterInitialization方法，在ApplicationContextAwareProcessor中并没有做过多逻辑处理。

```
public Object postProcessAfterInitialization(Object bean, String beanName) {
        return bean;
```

}

那么，我们重点看一下postProcessBeforeInitialization方法。

```
public Object postProcessBeforeInitialization(final Object bean, String
beanName) throws
BeansException {
    AccessControlContext acc = null;
    if (System.getSecurityManager() != null &&
        (bean instanceof EnvironmentAware || bean instanceof
EmbeddedValue
        ResolverAware ||
            bean instanceof ResourceLoaderAware || bean instanceof
            ApplicationEventPublisherAware ||
            bean instanceof MessageSourceAware || bean instanceof
            ApplicationContextAware)) {
        acc =
this.applicationContext.getBeanFactory().getAccessControlContext();
    }
    if (acc != null) {
        AccessController.doPrivileged(new PrivilegedAction<Object>() {
            public Object run() {
                invokeAwareInterfaces(bean);
                return null;
            }
        }, acc);
    }
    else {
        invokeAwareInterfaces(bean);
```

```
        }
        return bean;
    }
    private void invokeAwareInterfaces(Object bean) {
        if (bean instanceof Aware) {
            if (bean instanceof EnvironmentAware) {
                ((EnvironmentAware)
bean).setEnvironment(this.applicationContext.
                getEnvironment());
            }
            if (bean instanceof EmbeddedValueResolverAware) {
                ((EmbeddedValueResolverAware)
bean).setEmbeddedValueResolver(
                new EmbeddedValueResolver(this.applicationContext. getBean
                Factory()));
            }
            if (bean instanceof ResourceLoaderAware) {
                ((ResourceLoaderAware)
bean).setResourceLoader(this.applicationContext);
            }
            if (bean instanceof ApplicationEventPublisherAware) {
                ((ApplicationEventPublisherAware)
bean).setApplicationEventPublisher
                (this.applicationContext);
            }
            if (bean instanceof MessageSourceAware) {
                ((MessageSourceAware)
```

```
bean).setMessageSource(this.applicationContext);
            }
            if (bean instanceof ApplicationContextAware) {
                ((ApplicationContextAware) bean).setApplicationContext(this.
                applicationContext);
            }
        }
    }
```

postProcessBeforeInitialization 方法中调用了invokeAwareInterfaces。从 invokeAwareInterfaces方法中，我们或许已经或多或少了解了Spring的用意，实现这些Aware接口的bean在被初始化之后，可以取得一些对应的资源。

6.5.4 设置忽略依赖

当Spring将ApplicationContextAwareProcessor注册后，那么在invokeAwareInterfaces方法中间接调用的 Aware 类已经不是普通的 bean 了，如 ResourceLoaderAware、ApplicationEventPublisher Aware等，那么当然需要在Spring做bean的依赖注入的时候忽略它们。而ignoreDependencyInterface的作用正是在此。

```
//设置了几个忽略自动装配的接口

beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);

beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.cla

beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
```

beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);
beanFactory.ignoreDependencyInterface(EnvironmentAware.class);

6.5.5 注册依赖

Spring中有了忽略依赖的功能，当然也必不可少地会有注册依赖的功能。

beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
beanFactory.registerResolvableDependency(ResourceLoader.class, this);

beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);

beanFactory.registerResolvableDependency(ApplicationContext.class, this);

当注册了依赖解析后，例如当注册了对BeanFactory.class的解析依赖后，当bean的属性注入的时候，一旦检测到属性为BeanFactory类型便会将beanFactory的实例注入进去。

6.6 BeanFactory的后处理

BeanFacotry作为Spring中容器功能的基础，用于存放所有已经加载的bean，为了保证程序上的高可扩展性，Spring 针对 BeanFactory 做了大量的扩展，比如我们熟知的 PostProcessor等都是在这里实现的。

6.6.1 激活注册的BeanFactoryPostProcessor

正式开始介绍之前我们先了解下 BeanFactoryPostProcessor的用法。

BeanFactoryPostProcessor接口跟BeanPostProcessor类似，可以对

bean的定义（配置元数据）进行处理。也就是说，Spring IoC容器允许BeanFactoryPostProcessor在容器实际实例化任何其他的 bean 之前读取配置元数据，并有可能修改它。如果你愿意，你可以配置多个BeanFactoryPostProcessor。你还能通过设置“order”属性来控制BeanFactoryPostProcessor的执行次序（仅当BeanFactoryPostProcessor 实现了Ordered 接口时你才可以设置此属性，因此在实现BeanFactoryPostProcessor 时，就应当考虑实现 Ordered 接口）。请参考BeanFactoryPostProcessor和Ordered接口的JavaDoc以获取更详细的信息。

如果你想改变实际的 bean 实例（例如从配置元数据创建的对象），那么你最好使用BeanPostProcessor。同样地，BeanFactoryPostProcessor 的作用域范围是容器级的。它只和你所使用的容器有关。如果你在容器中定义一个 BeanFactoryPostProcessor，它仅仅对此容器中的bean进行后置处理。BeanFactoryPostProcessor不会对定义在另一个容器中的bean进行后置处理，即使这两个容器都是在同一层次上。在 Spring 中存在对于 BeanFactoryPostProcessor 的典型应用，比如PropertyPlaceholderConfigurer。

1．BeanFactoryPostProcessor的典型应用：**PropertyPlaceholderConfigurer**

有时候，阅读Spring的Bean描述文件时，你也许会遇到类似如下的一些配置：

```
<bean id="message" class="distConfig.HelloMessage">
    <property name="mes">
      <value>${bean.message}</value>
   </property>
</bean>
```

其中竟然出现了变量引用：${bean.message}。这就是Spring的分散

配置，可以在另外的配置文件中为bean.message指定值。如在bean.property配置如下定义：

bean.message=Hi,can you find me?

当访问名为message的bean时，mes属性就会被置为字符串“ Hi,can you find me?”，但Spring框架是怎么知道存在这样的配置文件呢？这就要靠PropertyPlaceholderConfigurer这个类的bean：

```
<bean id="mesHandler"
class="org.Springframework.beans.factory.config. Property Placeholder
Configurer">
    <property name="locations">
      <list>
        <value>config/bean.properties</value>
      </list>
    </property>
</bean>
```

在这个bean中指定了配置文件为config/bean.properties。到这里似乎找到问题的答案了，但是其实还有个问题。这个“mesHandler”只不过是Spring框架管理的一个bean，并没有被别的bean或者对象引用，Spring的beanFactory是怎么知道要从这个bean中获取配置信息的呢？

查看层级结构可以看出 PropertyPlaceholderConfigurer 这个类间接继承了 BeanFactory PostProcessor接口。这是一个很特别的接口，当Spring加载任何实现了这个接口的bean的配置时，都会在 bean 工厂载入所有 bean 的配置之后执行 postProcessBeanFactory 方法。在 PropertyResourceConfigurer 类中实现了 postProcessBeanFactory 方法，在方法中先后调用了mergeProperties、convertProperties、processProperties这3个方法，分别得到配置，将得到的配置转换为合适的类型，最后将配置内容告知BeanFactory。

正是通过实现BeanFactoryPostProcessor接口，BeanFactory会在实例化任何bean之前获得配置信息，从而能够正确解析bean描述文件中的变量引用。

2．使用自定义**BeanFactoryPostProcessor**

我们以实现一个BeanFactoryPostProcessor，去除潜在的“流氓”属性值的功能来展示自定义BeanFactoryPostProcessor的创建及使用，例如bean定义中留下bollocks这样的字眼。

配置文件BeanFactory.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.Springframework.org/schema/beans
  http://www.Springframework.org/schema/beans/Spring-beans.xsd
">
<bean id="bfpp"
class="com.Spring.ch04.ObscenityRemovingBeanFactoryPostProcessor">
    <property name="obscenties">
      <set>
        <value>bollocks</value>
        <value>winky</value>
        <value>bum</value>
        <value>Microsoft</value>
      </set>
    </property>
</bean>
<bean id="simpleBean"
class="com.Spring.ch04.SimplePostProcessor">
```

```
            <property name="connectionString" value="bollocks"/>
            <property name="password" value="imaginecup"/>
            <property name="username" value="Microsoft"/>
        </bean>
          </beans>
```

ObscenityRemovingBeanFactoryPostProcessor.java

```
public class ObscenityRemovingBeanFactoryPostProcessor implements
    BeanFactoryPostProcessor {
    private Set<String> obscenties;
    public ObscenityRemovingBeanFactoryPostProcessor(){
        this.obscenties=new HashSet<String>();
    }
    public void postProcessBeanFactory(
        ConfigurableListableBeanFactory beanFactory) throws
BeansException {
        String[] beanNames=beanFactory.getBeanDefinitionNames();
        for(String beanName:beanNames){
            BeanDefinition bd=beanFactory.getBeanDefinition(beanName);
            StringValueResolver valueResover=new StringValueResolver()
{
                public String resolveStringValue(String strVal) {
                    if(isObscene(strVal)) return "*****";
                    return strVal;
                }
            };
            BeanDefinitionVisitor visitor=new
BeanDefinitionVisitor(valueResover);
```

```
            visitor.visitBeanDefinition(bd);
        }
    }
    public boolean isObscene(Object value){
        String potentialObscenity=value.toString().toUpperCase();
        return this.obscenties.contains(potentialObscenity);
    }
    public void setObscenties(Set<String> obscenties) {
        this.obscenties.clear();
        for(String obscenity:obscenties){
            this.obscenties.add(obscenity.toUpperCase());
        }
    }
}
```

执行类：

```
public class PropertyConfigurerDemo {
    public static void main(String[] args) {
        ConfigurableListableBeanFactory bf=new XmlBeanFactory(new ClassPathResource
        ("/META-INF/BeanFactory.xml"));
        BeanFactoryPostProcessor bfpp=(BeanFactoryPostProcessor)bf.getBean("bfpp");
        bfpp.postProcessBeanFactory(bf);
        System.out.println(bf.getBean("simpleBean"));
    }
}
```

输出结果：

SimplePostProcessor{connectionString=*****,username=*****,passwo

通过ObscenityRemovingBeanFactoryPostProcessor Spring很好地实现了屏蔽掉obscenties定义的不应该展示的属性。

3．激活BeanFactoryPostProcessor

了解了BeanFactoryPostProcessor的用法后便可以深入研究BeanFactoryPostProcessor 的调用过程了。

```
protected void
invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory
beanFactory) {
        // Invoke BeanDefinitionRegistryPostProcessors first, if any.
        Set<String> processedBeans = new HashSet<String>();
       //对BeanDefinitionRegistry类型的处理
        if (beanFactory instanceof BeanDefinitionRegistry) {
            BeanDefinitionRegistry registry = (BeanDefinitionRegistry)
beanFactory;
            List<BeanFactoryPostProcessor> regularPostProcessors = new
LinkedList<BeanFactoryPostProcessor>();
           /**
             * BeanDefinitionRegistryPostProcessor
             */
            List<BeanDefinitionRegistryPostProcessor>
registryPostProcessors =
new LinkedList<BeanDefinitionRegistryPostProcessor>();
           /*
             * 硬编码注册的后处理器
             */
            for (BeanFactoryPostProcessor postProcessor :
```

```
getBeanFactoryPostProcessors()) {
                    if (postProcessor instanceof
BeanDefinitionRegistryPostProcessor) {
                        BeanDefinitionRegistryPostProcessor registryPostProcessor =
(Bean DefinitionRegistryPostProcessor) postProcessor;
                       //对于BeanDefinitionRegistryPostProcessor类型，在
BeanFactoryPostProcessor的基础上还有自己定义的方法，需要先调用

registryPostProcessor.postProcessBeanDefinitionRegistry(registry);
                       registryPostProcessors.add(registryPostProcessor);
                    }else {
                       //记录常规BeanFactoryPostProcessor
                       regularPostProcessors.add(postProcessor);
                   }
               }
               /*
                   * 配置注册的后处理器
                   */
                Map<String, BeanDefinitionRegistryPostProcessor> beanMap =
beanFactory.
                getBeansOfType(BeanDefinitionRegistryPostProcessor.class, true,
false);
                List<BeanDefinitionRegistryPostProcessor>
registryPostProcessorBeans =
                    new ArrayList<BeanDefinitionRegistryPostProcessor>
(beanMap.values());
               OrderComparator.sort(registryPostProcessorBeans);
```

```
                for (BeanDefinitionRegistryPostProcessor postProcessor :
registryPostProcessorBeans) {
                   //BeanDefinitionRegistryPostProcessor的特殊处理
                   postProcessor.postProcessBeanDefinitionRegistry(registry);
               }
               //激活postProcessBeanFactory方法，之前激活的是
postProcessBeanDefinitionRegistry
               //硬编码设置的BeanDefinitionRegistryPostProcessor
                invokeBeanFactoryPostProcessors(registryPostProcessors,
beanFactory);
               //配置的BeanDefinitionRegistryPostProcessor
                invokeBeanFactoryPostProcessors(registryPostProcessorBeans,
beanFactory);
               //常规BeanFactoryPostProcessor
                invokeBeanFactoryPostProcessors(regularPostProcessors,
beanFactory);
               processedBeans.addAll(beanMap.keySet());
           }
            else {
                // Invoke factory processors registered with the context instance.

invokeBeanFactoryPostProcessors(getBeanFactoryPostProcessors(),
beanFactory);
           }
           //对于配置中读取的BeanFactoryPostProcessor的处理
            String[] postProcessorNames =
beanFactory.getBeanNamesForType(BeanFactoryPost
```

```
            Processor.class, true, false);
            List<BeanFactoryPostProcessor> priorityOrderedPostProcessors =
new ArrayList
           <BeanFactoryPostProcessor>();
            List<String> orderedPostProcessorNames = new ArrayList<String>
();
            List<String> nonOrderedPostProcessorNames = new
ArrayList<String>();
           //对后处理器进行分类
            for (String ppName : postProcessorNames) {
                if (processedBeans.contains(ppName)) {
                   //已经处理过
                }else if (isTypeMatch(ppName, PriorityOrdered.class)) {

priorityOrderedPostProcessors.add(beanFactory.getBean(ppName,
                   BeanFactoryPostProcessor.class));
                }else if (isTypeMatch(ppName, Ordered.class)) {
                   orderedPostProcessorNames.add(ppName);
                }else {
                   nonOrderedPostProcessorNames.add(ppName);
               }
           }
           //按照优先级进行排序
           OrderComparator.sort(priorityOrderedPostProcessors);
            invokeBeanFactoryPostProcessors(priorityOrderedPostProcessors,
beanFactory);
            // Next, invoke the BeanFactoryPostProcessors that implement
```

```
Ordered.
     List<BeanFactoryPostProcessor> orderedPostProcessors = new
ArrayList<BeanFactory
    PostProcessor>();
     for (String postProcessorName : orderedPostProcessorNames) {
       orderedPostProcessors.add(getBean(postProcessorName,
BeanFactoryPostProcessor.
      class));
    }
    //按照order排序
    OrderComparator.sort(orderedPostProcessors);
     invokeBeanFactoryPostProcessors(orderedPostProcessors,
beanFactory);
    //无序，直接调用
     List<BeanFactoryPostProcessor> nonOrderedPostProcessors = new
ArrayList<BeanFactory
    PostProcessor>();
     for (String postProcessorName : nonOrderedPostProcessorNames) {
      nonOrderedPostProcessors.add(getBean(postProcessorName,
      BeanFactoryPostProcessor.class));
    }
     invokeBeanFactoryPostProcessors(nonOrderedPostProcessors,
beanFactory);
  }
```

从上面的方法中我们看到，对于 BeanFactoryPostProcessor 的处理主要分两种情况进行，一个是对于BeanDefinitionRegistry类的特殊处理，另一种是对普通的BeanFactoryPostProcessor进行处理。而对于每种

情况都需要考虑硬编码注入注册的后处理器以及通过配置注入的后处理器。

对于BeanDefinitionRegistry类型的处理类的处理主要包括以下内容。

（1）对于硬编码注册的后处理器的处理，主要是通过AbstractApplicationContext中的添加处理器方法addBeanFactoryPostProcessor进行添加。

```
public void addBeanFactoryPostProcessor(BeanFactoryPostProcessor beanFactoryPostProcessor) {
    this.beanFactoryPostProcessors.add(beanFactoryPostProcessor);
}
```

添加后的后处理器会存放在beanFactoryPostProcessors中，而在处理BeanFactoryPostProcessor时候会首先检测beanFactoryPostProcessors是否有数据。当然，BeanDefinitionRegistryPostProcessor继承自BeanFactoryPostProcessor，不但有BeanFactoryPostProcessor的特性，同时还有自己定义的个性化方法，也需要在此调用。所以，这里需要从beanFactoryPostProcessors 中挑出BeanDefinitionRegistryPostProcessor的后处理器，并进行其postProcessBeanDefinitionRegistry方法的激活。

（2）记录后处理器主要使用了三个List完成。

registryPostProcessors：记录通过硬编码方式注册的BeanDefinitionRegistryPostProcessor类型的处理器。

regularPostProcessors：记录通过硬编码方式注册的BeanFactoryPostProcessor类型的处理器。

registryPostProcessorBeans：记录通过配置方式注册的BeanDefinitionRegistryPostProcessor类型的处理器。

（3）对以上所记录的 List 中的后处理器进行统一调用BeanFactoryPostProcessor 的postProcessBeanFactory方法。

（4）对beanFactoryPostProcessors中非BeanDefinitionRegistryPostProcessor类型的后处理器进行统一的BeanFactoryPostProcessor的postProcessBeanFactory方法调用。

（5）普通beanFactory处理。

BeanDefinitionRegistryPostProcessor只对BeanDefinitionRegistry类型的ConfigurableListable BeanFactory有效，所以如果判断所示的beanFactory并不是BeanDefinitionRegistry，那么便可以忽略BeanDefinitionRegistryPostProcessor，而直接处理BeanFactoryPostProcessor，当然获取的方式与上面的获取类似。

这里需要提到的是，对于硬编码方式手动添加的后处理器是不需要做任何排序的，但是在配置文件中读取的处理器，Sping 并不保证读取的顺序。所以，为了保证用户的调用顺序的要求，Spring对于后处理器的调用支持按照PriorityOrdered或者Ordered的顺序调用。

6.6.2 注册BeanPostProcessor

上文中提到了BeanFacotoryPostProcessors的调用，现在我们来探索下BeanPostProcessor，但是这里并不是调用，而是注册。真正的调用其实是在 bean 的实例化阶段进行的。这是一个很重要的步骤，也是很多功能BeanFactory不支持的重要原因。Spring中大部分功能都是通过后处理器的方式进行扩展的，这是Spring框架的一个特性，但是在BeanFactory中其实并没有实现后处理器的自动注册，所以在调用的时候如果没有进行手动注册其实是不能使用的。但是在ApplicationContext中却添加了自动注册功能，如自定义这样一个后处理器：

```
public class MyInstantiationAwareBeanPostProcessor implements InstantiationAwareBean
PostProcessor{
    public Object postProcessBeforeInitialization(Object bean, String
```

beanName)
                throws BeansException {
            System.out.println("====");
             return null;
        }
        ... ...
    }

在配置文件中添加配置：

<bean class="processors.MyInstantiationAwareBeanPostProcessor"/>

那么使用 BeanFactory 方式进行 Spring 的 bean 的加载时是不会有任何改变的，但是使用ApplicationContext方式获取bean的时候会在获取每个bean时打印出“====”，而这个特性就是在registerBeanPostProcessors方法中完成的。

我们继续探索registerBeanPostProcessors的方法实现。

```
protected void
registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory)
{
        String[] postProcessorNames =
beanFactory.getBeanNamesForType(BeanPostProcessor.class,
        true, false);
       /*
          * BeanPostProcessorChecker是一个普通的信息打印，可能会有
些情况，
          * 当Spring的配置中的后处理器还没有被注册就已经开始了
bean的初始化时
          * 便会打印出BeanPostProcessorChecker中设定的信息
          */
```

```
 int beanProcessorTargetCount =
beanFactory.getBeanPostProcessorCount() + 1 +
postProcessorNames.length;
 beanFactory.addBeanPostProcessor(new
BeanPostProcessorChecker(beanFactory,
beanProcessorTargetCount));
//使用PriorityOrdered保证顺序
 List<BeanPostProcessor> priorityOrderedPostProcessors = new
ArrayList<Bean
PostProcessor>();
//MergedBeanDefinitionPostProcessor
 List<BeanPostProcessor> internalPostProcessors = new
ArrayList<BeanPost
Processor>();
//使用Ordered保证顺序
 List<String> orderedPostProcessorNames = new ArrayList<String>
();
//无序BeanPostProcessor
 List<String> nonOrderedPostProcessorNames = new
ArrayList<String>();
 for (String ppName : postProcessorNames) {
    if (isTypeMatch(ppName, PriorityOrdered.class)) {
       BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
      priorityOrderedPostProcessors.add(pp);
       if (pp instanceof MergedBeanDefinitionPostProcessor) {
          internalPostProcessors.add(pp);
```

```
            }
        }else if (isTypeMatch(ppName, Ordered.class)) {
            orderedPostProcessorNames.add(ppName);
        }else {
            nonOrderedPostProcessorNames.add(ppName);
        }
    }
    //第一步，注册所有实现PriorityOrdered的BeanPostProcessor
    OrderComparator.sort(priorityOrderedPostProcessors);
    registerBeanPostProcessors(beanFactory,
priorityOrderedPostProcessors);
    //第二步，注册所有实现Ordered的BeanPostProcessor
    List<BeanPostProcessor> orderedPostProcessors = new
ArrayList<BeanPostProcessor>();
    for (String ppName : orderedPostProcessorNames) {
        BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
        orderedPostProcessors.add(pp);
        if (pp instanceof MergedBeanDefinitionPostProcessor) {
            internalPostProcessors.add(pp);
        }
    }
    OrderComparator.sort(orderedPostProcessors);
    registerBeanPostProcessors(beanFactory, orderedPostProcessors);
    //第三步，注册所有无序的BeanPostProcessor
    List<BeanPostProcessor> nonOrderedPostProcessors = new
ArrayList<BeanPostProcessor>();
```

```
for (String ppName : nonOrderedPostProcessorNames) {
    BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
   nonOrderedPostProcessors.add(pp);
    if (pp instanceof MergedBeanDefinitionPostProcessor) {
      internalPostProcessors.add(pp);
   }
}
 registerBeanPostProcessors(beanFactory,
nonOrderedPostProcessors);
//第四步，注册所有MergedBeanDefinitionPostProcessor类型的
BeanPostProcessor，并非
重复注册，
//在beanFactory.addBeanPostProcessor中会先移除已经存在的
BeanPostProcessor
OrderComparator.sort(internalPostProcessors);
 registerBeanPostProcessors(beanFactory, internalPostProcessors);
//添加ApplicationListener探测器
 beanFactory.addBeanPostProcessor(new
ApplicationListenerDetector());
}
```

配合源码以及注释，在 registerBeanPostProcessors 方法中所做的逻辑相信大家已经很清楚了，我们再做一下总结。

首先我们会发现，对于 BeanPostProcessor 的处理与 BeanFactoryPostProcessor 的处理极为相似，但是似乎又有些不一样的地方。经过反复的对比发现，对于 BeanFactoryPostProcessor的处理要区分两种情况，一种方式是通过硬编码方式的处理，另一种是通过配置文件

方式的处理。那么为什么在 BeanPostProcessor 的处理中只考虑了配置文件的方式而不考虑硬编码的方式呢？提出这个问题，还是因为读者没有完全理解两者实现的功能。对于 BeanFactoryPostProcessor的处理，不但要实现注册功能，而且还要实现对后处理器的激活操作，所以需要载入配置中的定义，并进行激活；而对于BeanPostProcessor并不需要马上调用，再说，硬编码的方式实现的功能是将后处理器提取并调用，这里并不需要调用，当然不需要考虑硬编码的方式了，这里的功能只需要将配置文件的BeanPostProcessor提取出来并注册进入beanFactory就可以了。

对于 beanFactory 的注册，也不是直接注册就可以的。在 Spring 中支持对于 BeanPost Processor的排序，比如根据PriorityOrdered进行排序、根据Ordered进行排序或者无序，而Spring在BeanPostProcessor的激活顺序的时候也会考虑对于顺序的问题而先进行排序。

这里可能有个地方读者不是很理解，对于 internalPostProcessors 中存储的后处理器也就是MergedBeanDefinitionPostProcessor类型的处理器，在代码中似乎是被重复调用了，如：

```
for (String ppName : postProcessorNames) {
    if (isTypeMatch(ppName, PriorityOrdered.class)) {
        BeanPostProcessor pp = beanFactory.getBean(ppName,
BeanPostProcessor.class);
        priorityOrderedPostProcessors.add(pp);
        if (pp instanceof MergedBeanDefinitionPostProcessor) {
            internalPostProcessors.add(pp);
        }
    }else if (isTypeMatch(ppName, Ordered.class)) {
        orderedPostProcessorNames.add(ppName);
    }else {
        nonOrderedPostProcessorNames.add(ppName);
```

}

}

其实不是，我们可以看看对于registerBeanPostProcessors方法的实现方式。

```
private void registerBeanPostProcessors(
ConfigurableListableBeanFactory beanFactory,
List<BeanPostProcessor>
postProcessors) {
    for (BeanPostProcessor postProcessor : postProcessors) {
        beanFactory.addBeanPostProcessor(postProcessor);
    }
}
public void addBeanPostProcessor(BeanPostProcessor
beanPostProcessor) {
    Assert.notNull(beanPostProcessor, "BeanPostProcessor must not be
null");
    this.beanPostProcessors.remove(beanPostProcessor);
    this.beanPostProcessors.add(beanPostProcessor);
    if (beanPostProcessor instanceof
InstantiationAwareBeanPostProcessor) {
        this.hasInstantiationAwareBeanPostProcessors = true;
    }
    if (beanPostProcessor instanceof
DestructionAwareBeanPostProcessor) {
        this.hasDestructionAwareBeanPostProcessors = true;
}
    }
```

}

可以看到，在 registerBeanPostProcessors 方法的实现中其实已经确保了 beanPostProcessor的唯一性，个人猜想，之所以选择在registerBeanPostProcessors 中没有进行重复移除操作或许是为了保持分类的效果，使逻辑更为清晰吧。

6.6.3 初始化消息资源

在进行这段函数的解析之前，我们同样先来回顾Spring国际化的使用方法。

假设我们正在开发一个支持多国语言的Web应用程序，要求系统能够根据客户端的系统的语言类型返回对应的界面：英文的操作系统返回英文界面，而中文的操作系统则返回中文界面——这便是典型的i18n国际化问题。对于有国际化要求的应用系统，我们不能简单地采用硬编码的方式编写用户界面信息、报错信息等内容，而必须为这些需要国际化的信息进行特殊处理。简单来说，就是为每种语言提供一套相应的资源文件，并以规范化命名的方式保存在特定的目录中，由系统自动根据客户端语言选择适合的资源文件。

“国际化信息”也称为“本地化信息”，一般需要两个条件才可以确定一个特定类型的本地化信息，它们分别是“语言类型”和“国家/地区的类型”。如中文本地化信息既有中国大陆地区的中文，又有中国台湾地区、中国香港地区的中文，还有新加坡地区的中文。Java 通过java.util.Locale类表示一个本地化对象，它允许通过语言参数和国家/地区参数创建一个确定的本地化对象。

java.util.Locale是表示语言和国家/地区信息的本地化类，它是创建国际化应用的基础。下面给出几个创建本地化对象的示例：

```
//① 带有语言和国家/地区信息的本地化对象
Locale locale1 = new Locale("zh","CN");
```

//② 只有语言信息的本地化对象

Locale locale2 = new Locale("zh");

//③ 等同于Locale("zh","CN")

Locale locale3 = Locale.CHINA;

//④ 等同于Locale("zh")

Locale locale4 = Locale.CHINESE;

//⑤ 获取本地系统默认的本地化对象

Locale locale 5= Locale.getDefault();

JDK 的 java.util 包中提供了几个支持本地化的格式化操作工具类：NumberFormat、DateFormat、MessageFormat，而在 Spring 中的国际化资源操作也无非是对于这些类的封装操作，我们仅仅介绍下MessageFormat的用法以帮助大家回顾：

//①信息格式化串

String pattern1 = "{0}，你好！你于{1}在工商银行存入{2} 元。";

String pattern2 = "At {1,time,short} On{1,date,long}，{0} paid {2,number, currency}.";

//②用于动态替换占位符的参数

Object[] params = {"John", new GregorianCalendar().getTime(),1.0E3};

//③使用默认本地化对象格式化信息

String msg1 = MessageFormat.format(pattern1,params);

//④使用指定的本地化对象格式化信息

MessageFormat mf = new MessageFormat(pattern2,Locale.US);

String msg2 = mf.format(params);

System.out.println(msg1);

System.out.println(msg2);

Spring 定义了访问国际化信息的 MessageSource 接口，并提供了几个易用的实现类。MessageSource分别被HierarchicalMessageSource和

ApplicationContext接口扩展，这里我们主要看一下HierarchicalMessageSource接口的几个实现类，如图6-3所示。

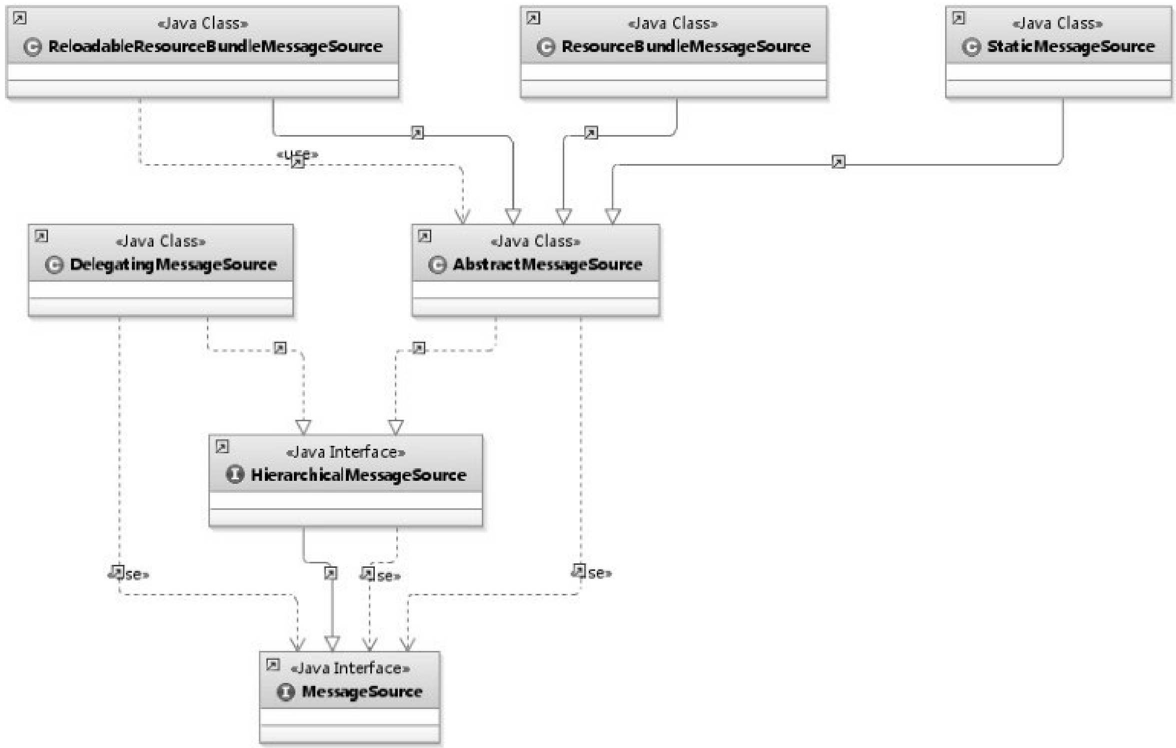


图6-3 MessageSource类图结构

HierarchicalMessageSource 接口最重要的两个实现类是ResourceBundleMessageSource 和ReloadableResourceBundleMessageSource。它们基于Java的ResourceBundle基础类实现，允许仅通过资源名加载国际化资源。ReloadableResourceBundleMessageSource提供了定时刷新功能，允许在不重启系统的情况下，更新资源的信息。StaticMessageSource主要用于程序测试，它允许通过编程的方式提供国际化信息。而DelegatingMessageSource是为方便操作父MessageSource而提供的代理类。仅仅举例ResourceBundleMessageSource的实现方式。

（1）定义资源文件。

messages.properties（默认：英文），内容仅一句，如下：

test=test

messages_zh_CN.properties（简体中文）：

test=测试

然后 cmd，打开命令行窗口，输入 native2ascii -encoding gbk C:\messages_zh_CN.properties C:\messages_zh_CN_tem.properties，并将 C:\messages_zh_CN_tem.properties 中的内容替换到 messages_zh_CN.properties 中，这样 messages_zh_CN.properties 文件就存放的是转码后的内容了，比较简单。

（2）定义配置文件。

```
<bean id="messageSource"
class="org.Springframework.context.support.ResourceBundleMessageSource"
      <property name="basenames">
        <list>
          <value>test/messages</value>
        </list>
      </property>
</bean>
```

其中，这个Bean的ID必须命名为messageSource，否则会抛出NoSuchMessageException异常。

（3）使用。通过ApplicationContext访问国际化信息。

```
String[] configs = {"applicationContext.xml"};
ApplicationContext ctx = new
ClassPathXmlApplicationContext(configs);
//①直接通过容器访问国际化信息
Object[] params = {"John", new GregorianCalendar().getTime()};
String str1 = ctx.getMessage("test",params,Locale.US);
```

String str2 = ctx.getMessage("test",params,Locale.CHINA);

System.out.println(str1);

System.out.println(str2);

了解了Spring国际化的使用后便可以进行源码的分析了。

在initMessageSource中的方法主要功能是提取配置中定义的messageSource，并将其记录在Spring的容器中，也就是AbstractApplicationContext中。当然，如果用户未设置资源文件的话，Spring中也提供了默认的配置DelegatingMessageSource。

在 initMessageSource 中获取自定义资源文件的方式为beanFactory.getBean(MESSAGE_ SOURCE_BEAN_NAME, MessageSource.class)，在这里Spring使用了硬编码的方式硬性规定了子定义资源文件必须为 message，否则便会获取不到自定义资源配置，这也是为什么之前提到Bean的id如果部位message会抛出异常。

protected void initMessageSource() {

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

if

(beanFactory.containsLocalBean(MESSAGE_SOURCE_BEAN_NAME)) {

//如果在配置中已经配置了 messageSource ，那么将messageSource 提取并记录在

this.messageSource中

this.messageSource =

beanFactory.getBean(MESSAGE_SOURCE_BEAN_NAME,

MessageSource.

class);

// Make MessageSource aware of parent MessageSource.

if (this.parent != null && this.messageSource instanceof

Hierarchical

```
MessageSource) {
    HierarchicalMessageSource hms =
(HierarchicalMessageSource) this.
   messageSource;
    if (hms.getParentMessageSource() == null) {

hms.setParentMessageSource(getInternalParentMessageSource());
   }
}
if (logger.isDebugEnabled()) {
    logger.debug("Using MessageSource [" + this.messageSource +
"]");
}
}else {
//如果用户并没有定义配置文件，那么使用临时的
DelegatingMessageSource以便于作为调用
getMessage方法的返回。
DelegatingMessageSource dms = new
DelegatingMessageSource();
dms.setParentMessageSource(getInternalParentMessageSource());
this.messageSource = dms;

beanFactory.registerSingleton(MESSAGE_SOURCE_BEAN_NAME,
this.messageSource);
if (logger.isDebugEnabled()) {
    logger.debug("Unable to locate MessageSource with name '" +
    MESSAGE_SOURCE_BEAN_NAME +
```

```
"': using default [" + this.messageSource + "]");
        }
    }
}
```

通过读取并将自定义资源文件配置记录在容器中，那么就可以在获取资源文件的时候直接使用了，例如，在AbstractApplicationContext中的获取资源文件属性的方法：

```
public String getMessage(String code, Object args[], Locale locale)
throws NoSuchMessage
Exception {
    return getMessageSource().getMessage(code, args, locale);
}
```

其中的getMessageSource()方法正是获取了之前定义的自定义资源配置。

6.6.4 初始化ApplicationEventMulticaster

在讲解Spring的时间传播器之前，我们还是先来看一下Spring的事件监听的简单用法。

（1）定义监听事件。

```
public class TestEvent extends ApplicationEvent {
public String msg;
public TestEvent (Object source) {
   super(source);
}
public TestEvent (Object source, String msg) {
   super(source);
    this.msg = msg;
```

```
}
public void print(){
   System.out.println(msg);
}
}
```

（2）定义监听器。

```
public class TestListener implements ApplicationListener {
public void onApplicationEvent(ApplicationEvent event) {
    if(event instanceof TestEvent){
       TestEvent testEvent = (TestEvent)event;
       testEvent .print();
   }
}
}
```

（3）添加配置文件。

```
<bean id="testListener" class="com.test.event.TestListener "/>
```

（4）测试。

```
public class Test {
    public static void main(String[] args) {
       ApplicationContext context = new
ClassPathXmlApplicationContext ("classpath:
      applicationContext.xml");
       TestEvent event = new TestEvent ("hello",”msg”);
      context.publishEvent(event);
   }
}
```

当程序运行时，Spring会将发出的TestEvent事件转给我们自定义的

TestListener 进行进一步处理。

或许很多人一下子会反映出设计模式中的观察者模式，这确实是个典型的应用，可以在比较关心的事件结束后及时处理。那么我们看看ApplicationEventMulticaster是如何被初始化的，以确保功能的正确运行。

initApplicationEventMulticaster的方式比较简单，无非考虑两种情况。

如果用户自定义了事件广播器，那么使用用户自定义的事件广播器。

如果用户没有自定义事件广播器，那么使用默认的ApplicationEventMulticaster。

```
protected void initApplicationEventMulticaster() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if
(beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER
{
        this.applicationEventMulticaster =

beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_N
            ApplicationEventMulticaster.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using ApplicationEventMulticaster [" +
this.application
            EventMulticaster + "]");
        }
    }else {
        this.applicationEventMulticaster = new
```

```
SimpleApplicationEventMulticaster
                (beanFactory);

beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_B
                this.applicationEventMulticaster);
                 if (logger.isDebugEnabled()) {
                     logger.debug("Unable to locate ApplicationEventMulticaster
with name '" +
                     APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
                     "': using default [" + this.applicationEventMulticaster + "]");
                }
            }
        }
```

按照之前介绍的顺序及逻辑，我们推断，作为广播器，一定是用于存放监听器并在合适的时候调用监听器，那么我们不妨进入默认的广播器实现SimpleApplicationEventMulticaster来一探究竟。

其中的一段代码是我们感兴趣的。

```
        public void multicastEvent(final ApplicationEvent event) {
             for (final ApplicationListener listener :
getApplicationListeners(event)) {
                 Executor executor = getTaskExecutor();
                 if (executor != null) {
                     executor.execute(new Runnable() {
                        @SuppressWarnings("unchecked")
                         public void run() {
                        listener.onApplicationEvent(event);
                        }
```

```
            });
        }
        else {
            listener.onApplicationEvent(event);
        }
    }
}
```

可以推断，当产生 Spring 事件的时候会默认使用 SimpleApplicationEventMulticaster 的multicastEvent来广播事件，遍历所有监听器，并使用监听器中的onApplicationEvent方法来进行监听器的处理。而对于每个监听器来说其实都可以获取到产生的事件，但是是否进行处理则由事件监听器来决定。

6.6.5 注册监听器

之前在介绍 Spring 的广播器时反复提到了事件监听器，那么在 Spring 注册监听器的时候又做了哪些逻辑操作呢？

```
protected void registerListeners() {
    //硬编码方式注册的监听器处理
     for (ApplicationListener<?> listener : getApplicationListeners()) {
        getApplicationEventMulticaster().addApplicationListener(listener);
    }
    //配置文件注册的监听器处理
     String[] listenerBeanNames =
getBeanNamesForType(ApplicationListener.class,
     true, false);
     for (String lisName : listenerBeanNames) {
```

```
getApplicationEventMulticaster().addApplicationListenerBean(lisName);
    }
  }
```

6.7 初始化非延迟加载单例

完成BeanFactory的初始化工作，其中包括ConversionService的设置、配置冻结以及非延迟加载的bean的初始化工作。

```
protected void
finishBeanFactoryInitialization(ConfigurableListableBeanFactory
beanFactory) {
    // Initialize conversion service for this context.
    if
(beanFactory.containsBean(CONVERSION_SERVICE_BEAN_NAME) &&

beanFactory.isTypeMatch(CONVERSION_SERVICE_BEAN_NAME,
      ConversionService.class)) {
    beanFactory.setConversionService(
      beanFactory.getBean(CONVERSION_SERVICE_BEAN_NAME,
      ConversionService.class));
    }
    // Initialize LoadTimeWeaverAware beans early to allow for
registering their
    transformers early.
    String[] weaverAwareNames = beanFactory.getBeanNamesForType
(LoadTimeWeaverAware.
    class, false, false);
```

```
        for (String weaverAwareName : weaverAwareNames) {
            getBean(weaverAwareName);
        }
        // Stop using the temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(null);
        //冻结所有的bean定义，说明注册的bean定义将不被修改或任何进一步的处理。
        beanFactory.freezeConfiguration();
        // Instantiate all remaining (non-lazy-init) singletons.
        //初始化剩下的单实例（非惰性的）
        beanFactory.preInstantiateSingletons();
    }
```

首先我们来了解一下ConversionService类所提供的作用。

1．ConversionService的设置

之前我们提到过使用自定义类型转换器从String转换为Date的方式，那么，在Spring中还提供了另一种转换方式：使用Converter。同样，我们使用一个简单的示例来了解下Converter的使用方式。

（1）定义转换器。

```
public class String2DateConverter implements Converter<String, Date> {
    @Override
    public Date convert(String arg0) {
        try {
            return DateUtils.parseDate(arg0,
                new String[] { "yyyy-MM-dd HH:mm:ss" });
        } catch (ParseException e) {
            return null;
```

```
        }
    }
}
```

（2）注册。

```
<bean id="conversionService"

class="org.Springframework.context.support.ConversionServiceFactoryBean'
    <property name="converters">
      <list>
          <bean class="String2DateConverter" />
      </list>
    </property>
</bean>
```

（3）测试。

这样便可以使用 Converter 为我们提供的功能了，下面我们通过一个简便的方法来对此直接测试。

```
public void testStringToPhoneNumberConvert() {
    DefaultConversionService conversionService = new
DefaultConversionService();
    conversionService.addConverter(new
StringToPhoneNumberConverter());
    String phoneNumberStr = "010-12345678";
    PhoneNumberModel phoneNumber =
conversionService.convert(phoneNumberStr, PhoneNumber
    Model.class);
    Assert.assertEquals("010", phoneNumber.getAreaCode());
}
```

通过以上的功能我们看到了Converter以及ConversionService提供的便利功能，其中的配置就是在当前函数中被初始化的。

2．冻结配置

冻结所有的bean定义，说明注册的bean定义将不被修改或进行任何进一步的处理。

```
public void freezeConfiguration() {
    this.configurationFrozen = true;
    synchronized (this.beanDefinitionMap) {
        this.frozenBeanDefinitionNames =
StringUtils.toStringArray(this.bean
        DefinitionNames);
    }
}
```

3．初始化非延迟加载

ApplicationContext 实现的默认行为就是在启动时将所有单例 bean 提前进行实例化。提前实例化意味着作为初始化过程的一部分，ApplicationContext实例会创建并配置所有的单例bean。通常情况下这是一件好事，因为这样在配置中的任何错误就会即刻被发现（否则的话可能要花几个小时甚至几天）。而这个实例化的过程就是在finishBeanFactoryInitialization 中完成的。

```
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isInfoEnabled()) {
        this.logger.info("Pre-instantiating singletons in " + this);
    }
    List<String> beanNames;
    synchronized (this.beanDefinitionMap) {
        // Iterate over a copy to allow for init methods which in turn
```

```
register new
                bean definitions.
                // While this may not be part of the regular factory bootstrap, it
does
                otherwise work fine.
                beanNames = new ArrayList<String>(this.beanDefinitionNames);
            }
            for (String beanName : beanNames) {
                RootBeanDefinition bd =
getMergedLocalBeanDefinition(beanName);
                if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
                    if (isFactoryBean(beanName)) {
                        final FactoryBean<?> factory = (FactoryBean<?>)
getBean(FACTORY_
                       BEAN_PREFIX + beanName);
                        boolean isEagerInit;
                        if (System.getSecurityManager() != null && factory
instanceof
                        SmartFactoryBean) {
                            isEagerInit = AccessController.doPrivileged(new
PrivilegedAction
                            <Boolean>() {
                                public Boolean run() {
                                    return ((SmartFactoryBean<?>) factory).isEagerInit();
                                }
                            }, getAccessControlContext());
                       }
```

```
            else {
                isEagerInit = (factory instanceof SmartFactoryBean &&
                ((SmartFactoryBean<?>) factory).isEagerInit());
            }
            if (isEagerInit) {
               getBean(beanName);
            }
        }
        else {
           getBean(beanName);
        }
    }
  }
}
```

6.8 finishRefresh

在Spring中还提供了Lifecycle接口，Lifecycle中包含start/stop方法，实现此接口后Spring会保证在启动的时候调用其start方法开始生命周期，并在Spring关闭的时候调用stop方法来结束生命周期，通常用来配置后台程序，在启动后一直运行（如对MQ进行轮询等）。而ApplicationContext的初始化最后正是保证了这一功能的实现。

```
protected void finishRefresh() {
    initLifecycleProcessor();
     // Propagate refresh to lifecycle processor first.
    getLifecycleProcessor().onRefresh();
     // Publish the final event.
```

publishEvent(new ContextRefreshedEvent(this));
}

1．initLifecycleProcessor

当ApplicationContext启动或停止时，它会通过LifecycleProcessor来与所有声明的bean的周期做状态更新，而在LifecycleProcessor的使用前首先需要初始化。

```
protected void initLifecycleProcessor() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if
(beanFactory.containsLocalBean(LIFECYCLE_PROCESSOR_BEAN_NAM
{
        this.lifecycleProcessor =

beanFactory.getBean(LIFECYCLE_PROCESSOR_BEAN_NAME,
            LifecycleProcessor.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using LifecycleProcessor [" +
this.lifecycleProcessor + "]");
        }
    }else {
        DefaultLifecycleProcessor defaultProcessor = new
DefaultLifecycleProcessor();
        defaultProcessor.setBeanFactory(beanFactory);
        this.lifecycleProcessor = defaultProcessor;

beanFactory.registerSingleton(LIFECYCLE_PROCESSOR_BEAN_NAME,
        this.lifecycleProcessor);
```

```
            if (logger.isDebugEnabled()) {
                logger.debug("Unable to locate LifecycleProcessor with name '"
+
                    LIFECYCLE_PROCESSOR_BEAN_NAME +
                    "': using default [" + this.lifecycleProcessor + "]");
            }
        }
    }
```

2. onRefresh

启动所有实现了Lifecycle接口的bean。

```
    public void onRefresh() {
        startBeans(true);
         this.running = true;
    }
    private void startBeans(boolean autoStartupOnly) {
         Map<String, Lifecycle> lifecycleBeans = getLifecycleBeans();
         Map<Integer, LifecycleGroup> phases = new HashMap<Integer,
LifecycleGroup>();
         for (Map.Entry<String, ? extends Lifecycle> entry :
lifecycleBeans.entrySet()) {
             Lifecycle bean = entry.getValue();
             if (!autoStartupOnly || (bean instanceof SmartLifecycle &&
((SmartLifecycle)
             bean).isAutoStartup())) {
                 int phase = getPhase(bean);
                 LifecycleGroup group = phases.get(phase);
                 if (group == null) {
```

```
                group = new LifecycleGroup(phase,
this.timeoutPerShutdownPhase,
                lifecycleBeans, autoStartupOnly);
                phases.put(phase, group);
            }
            group.add(entry.getKey(), bean);
        }
    }
    if (phases.size() > 0) {
        List<Integer> keys = new ArrayList<Integer>(phases.keySet());
        Collections.sort(keys);
        for (Integer key : keys) {
            phases.get(key).start();
        }
    }
}
```

3．publishEvent

当完成ApplicationContext初始化的时候，要通过Spring中的事件发布机制来发出Context RefreshedEvent事件，以保证对应的监听器可以做进一步的逻辑处理。

```
public void publishEvent(ApplicationEvent event) {
    Assert.notNull(event, "Event must not be null");
    if (logger.isTraceEnabled()) {
        logger.trace("Publishing event in " + getDisplayName() + ": " +
event);
    }
    getApplicationEventMulticaster().multicastEvent(event);
```

```
        if (this.parent != null) {
            this.parent.publishEvent(event);
        }
    }
```

第7章 AOP

我们知道，使用面向对象编程（OOP）有一些弊端，当需要为多个不具有继承关系的对象引入同一个公共行为时，例如日志、安全检测等，我们只有在每个对象里引用公共行为，这样程序中就产生了大量的重复代码，程序就不便于维护了，所以就有了一个对面向对象编程的补充，即面向方面编程（AOP），AOP所关注的方向是横向的，不同于OOP的纵向。

Spring中提供了AOP的实现，但是在低版本Spring中定义一个切面是比较麻烦的，需要实现特定的接口，并进行一些较为复杂的配置。低版本 Spring AOP的配置是被批评最多的地方。Spring听取了这方面的批评声音，并下决心彻底改变这一现状。在Spring 2.0中，Spring AOP已经焕然一新，你可以使用@AspectJ注解非常容易地定义一个切面，不需要实现任何的接口。

Spring 2.0采用@AspectJ注解对POJO进行标注，从而定义一个包含切点信息和增强横切逻辑的切面。Spring 2.0可以将这个切面织入到匹配的目标Bean中。@AspectJ注解使用AspectJ切点表达式语法进行切点定义，可以通过切点函数、运算符、通配符等高级功能进行切点定义，拥有强大的连接点描述能力。我们先来直观地浏览一下Spring中的AOP实现。

7.1 动态AOP使用示例

（1）创建用于拦截的bean。

在实际工作中，此bean可能是满足业务需要的核心逻辑，例如test方法中可能会封装着某个核心业务，但是，如果我们想在test前后加入日志来跟踪调试，如果直接修改源码并不符合面向对象的设计方法，而且随意改动原有代码也会造成一定的风险，还好接下来的Spring帮我们做到了这一点。

```
public class TestBean{
    private String testStr = "testStr";
    public String getTestStr() {
        return testStr;
    }
    public void setTestStr(String testStr) {
        this.testStr = testStr;
    }
    public void test(){
        System.out.println("test");
    }
}
```

（2）创建Advisor。

Spring中摒弃了最原始的繁杂配置方式而采用 @AspectJ注解对POJO进行标注，使AOP的工作大大简化，例如，在AspectJTest类中，我们要做的就是在所有类的test方法执行前在控制台中打印beforeTest，而在所有类的test方法执行后打印afterTest，同时又使用环绕的方式在所有类的方法执行前后再次分别打印before1和after1。

```
@Aspect
public class AspectJTest {
   @Pointcut("execution(* *.test(..))")
```

```
 public void test(){
}
@Before("test()")
 public void beforeTest(){
    System.out.println("beforeTest");
}
@After("test()")
 public void afterTest(){
    System.out.println("afterTest");
}
@Around("test()")
 public Object arountTest(ProceedingJoinPoint p){
    System.out.println("before1");
     Object o=null;
     try {
         o = p.proceed();
     } catch (Throwable e) {
        e.printStackTrace();
    }
    System.out.println("after1");
     return o;
}
}
```

（3）创建配置文件。

XML是Spring的基础。尽管Spring一再简化配置，并且大有使用注解取代XML配置之势，但是无论如何，至少现在XML还是Spring的基础。要在Spring中开启AOP功能，还需要在配置文件中作如下声明：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.Springframework.org/schema/aop"
   xmlns:context="http://www.Springframework.org/schema/context"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
       http://www.Springframework.org/schema/beans/Spring-beans-3.0.xsd
       http://www.Springframework.org/schema/aop
       http://www.Springframework.org/schema/aop/Spring-aop-3.0.xsd
       http://www.Springframework.org/schema/context
        http://www.Springframework.org/schema/context/Spring- context-3.0.xsd
   ">
<aop:aspectj-autoproxy />
<bean id="test" class="test.TestBean"/>
<bean class="test.AspectJTest"/>
</beans>
```

（4）测试。

经过以上步骤后，便可以验证Spring的AOP为我们提供的神奇效果了。

```
public static void main(String[] args) {
    ApplicationContext bf = new ClassPathXmlApplicationContext("aspectTest.xml");
    TestBean bean=(TestBean) bf.getBean("test");
   bean.test();
```

}

不出意外，我们会看到控制台中打印了如下代码：

beforeTest

before1

test

afterTest

after1

Spring实现了对所有类的test方法进行增强，使辅助功能可以独立于核心业务之外，方便与程序的扩展和解耦。

那么，Spring究竟是如何实现AOP的呢？首先我们知道，Spring是否支持注解的AOP是由一个配置文件控制的，也就是<aop:aspectj-autoproxy />，当在配置文件中声明了这句配置的时候，Spring就会支持注解的AOP，那么我们的分析就从这句注解开始。

7.2 动态AOP自定义标签

之前讲过Spring中的自定义注解，如果声明了自定义的注解，那么就一定会在程序中的某个地方注册了对应的解析器。我们搜索整个代码，尝试找到注册的地方，全局搜索后我们发现了在AopNamespaceHandler中对应着这样一段函数：

```
public void init() {
    // In 2.0 XSD as well as in 2.1 XSD.
    registerBeanDefinitionParser("config", new
ConfigBeanDefinitionParser());
    registerBeanDefinitionParser("aspectj-autoproxy", new
AspectJAutoProxyBean
   DefinitionParser());
```

registerBeanDefinitionDecorator("scoped-proxy", new ScopedProxyBeanDefinition
Decorator());
// Only in 2.0 XSD: moved to context namespace as of 2.1
registerBeanDefinitionParser("Spring-configured", new SpringConfiguredBean
DefinitionParser());
}

此处不再对Spring中的自定义注解方式进行讨论。有兴趣的读者可以回顾之前的内容。

我们可以得知，在解析配置文件的时候，一旦遇到 aspectj-autoproxy 注解时就会使用解析器AspectJAutoProxyBeanDefinitionParser进行解析，那么我们来看一看 AspectJAutoProxyBean DefinitionParser的内部实现。

7.2.1 注册AnnotationAwareAspectJAutoProxyCreator

所有解析器，因为是对BeanDefinitionParser接口的统一实现，入口都是从parse函数开始的，AspectJAutoProxyBeanDefinitionParser的parse函数如下：

```
public BeanDefinition parse(Element element, ParserContext parserContext) {
    //注册AnnotationAwareAspectJAutoProxyCreator

AopNamespaceUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary
    (parserContext, element);
    //对于注解中子类的处理
    extendBeanDefinition(element, parserContext);
```

return null;

}

其中 registerAspectJAnnotationAutoProxyCreatorIfNecessary 函数是我们比较关心的，也是关键逻辑的实现。

```
/**
 * 注册AnnotationAwareAspectJAutoProxyCreator
 * @param parserContext
 * @param sourceElement
 */
public static void
registerAspectJAnnotationAutoProxyCreatorIfNecessary(
    ParserContext parserContext, Element sourceElement) {
    //注册或升级 AutoProxyCreator 定义 beanName 为
org.Springframework.aop.config.
    internalAutoProxyCreator的BeanDefinition
    BeanDefinition beanDefinition =
AopConfigUtils.registerAspectJAnnotationAuto
    ProxyCreatorIfNecessary(
    parserContext.getRegistry(),
parserContext.extractSource(sourceElement));
    //对于proxy-target-class以及expose-proxy属性的处理
    useClassProxyingIfNecessary(parserContext.getRegistry(),
sourceElement);
    //注册组件并通知，便于监听器做进一步处理
//其中beanDefinition的className为
AnnotationAwareAspectJAutoProxyCreator
    registerComponentIfNecessary(beanDefinition, parserContext);
```

}

在 registerAspectJAnnotationAutoProxyCreatorIfNecessary 方法中主要完成了 3 件事情，基本上每行代码就是一个完整的逻辑。

1．注册或者升级AnnotationAwareAspectJAutoProxyCreator

对于AOP的实现，基本上都是靠AnnotationAwareAspectJAutoProxyCreator去完成，它可以根据@Point 注解定义的切点来自动代理相匹配的 bean。但是为了配置简便，Spring 使用了自定义配置来帮助我们自动注册AnnotationAwareAspectJAutoProxyCreator，其注册过程就是在这里实现的。

```
public static BeanDefinition
registerAspectJAnnotationAutoProxyCreatorIfNecessary
(BeanDefinitionRegistry registry, Object source) {
    return
registerOrEscalateApcAsRequired(AnnotationAwareAspectJAutoProxyCreat
    class, registry, source);
}
private static BeanDefinition registerOrEscalateApcAsRequired(Class
cls, BeanDefinition
Registry registry, Object source) {
    Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
//如果已经存在了自动代理创建器且存在的自动代理创建器与现在的不一致那么需要根据优先级来判断到底需要使用哪
    if
(registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAM
{
    //AUTO_PROXY_CREATOR_BEAN_NAME =
```

```
//"org.Springframework.aop.config.internalAutoProxyCreator";
 BeanDefinition apcDefinition =
registry.getBeanDefinition(AUTO_PROXY_
CREATOR_BEAN_NAME);
 if (!cls.getName().equals(apcDefinition.getBeanClassName())) {
int currentPriority =
findPriorityForClass(apcDefinition.getBean ClassName());
 int requiredPriority = findPriorityForClass(cls);
 if (currentPriority < requiredPriority) {
  //改变bean最重要的就是改变bean所对应的className属性
  apcDefinition.setBeanClassName(cls.getName());
 }
}
 //如果已经存在自动代理创建器并且与将要创建的一致，那么
无需再此创建
 return null;
}
 RootBeanDefinition beanDefinition = new RootBeanDefinition(cls);
beanDefinition.setSource(source);
beanDefinition.getPropertyValues().add("order",
Ordered.HIGHEST_PRECEDENCE);

beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
 //AUTO_PROXY_CREATOR_BEAN_NAME =
//"org.Springframework.aop.config.internalAutoProxyCreator";

registry.registerBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAME
```

beanDefinition);
    return beanDefinition;
}

以上代码中实现了自动注册AnnotationAwareAspectJAutoProxyCreator类的功能，同时这里还涉及了一个优先级的问题，如果已经存在了自动代理创建器，而且存在的自动代理创建器与现在的不一致，那么需要根据优先级来判断到底需要使用哪个。

2．处理proxy-target-class以及expose-proxy属性

useClassProxyingIfNecessary实现了proxy-target-class属性以及expose-proxy属性的处理。

```
private static void
useClassProxyingIfNecessary(BeanDefinitionRegistry registry, Element
sourceElement) {
    if (sourceElement != null) {
      //对于proxy-target-class属性的处理。
       boolean proxyTargetClass =
Boolean.valueOf(sourceElement.getAttribute
      (PROXY_TARGET_CLASS_ATTRIBUTE));
       if (proxyTargetClass) {

AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
      }
      //对于expose-proxy属性的处理
       boolean exposeProxy =
Boolean.valueOf(sourceElement.getAttribute (EXPOSE_
      PROXY_ATTRIBUTE));
```

```
            if (exposeProxy) {
AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
            }
        }
    }
    //强制使用的过程其实也是一个属性设置的过程
    public static void
forceAutoProxyCreatorToUseClassProxying(BeanDefinitionRegistry
    registry) {
        if
(registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAM
{
            BeanDefinition definition =
registry.getBeanDefinition(AUTO_PROXY_CREATOR_
            BEAN_NAME);
            definition.getPropertyValues().add("proxyTargetClass",
Boolean.TRUE);
        }
    }
    static void
forceAutoProxyCreatorToExposeProxy(BeanDefinitionRegistry registry) {
        if
(registry.containsBeanDefinition(AUTO_PROXY_CREATOR_BEAN_NAM
{
            BeanDefinition definition =
registry.getBeanDefinition(AUTO_PROXY_CREATOR_
```

BEAN_NAME);
definition.getPropertyValues().add("exposeProxy", Boolean.TRUE);
}
}

proxy-target-class：Spring AOP部分使用 JDK动态代理或者CGLIB来为目标对象创建代理。（建议尽量使用JDK的动态代理），如果被代理的目标对象实现了至少一个接口，则会使用JDK动态代理。所有该目标类型实现的接口都将被代理。若该目标对象没有实现任何接口，则创建一个CGLIB代理。如果你希望强制使用CGLIB代理，（例如希望代理目标对象的所有方法，而不只是实现自接口的方法）那也可以。但是需要考虑以下两个问题。

◆无法通知（advise）Final方法，因为它们不能被覆写。

◆ 你需要将CGLIB二进制发行包放在classpath下面。

与之相较，JDK本身就提供了动态代理，强制使用 CGLIB代理需要将 <aop:config>的proxy-target-class属性设为true:

<aop:config proxy-target-class="true"> ... </aop:config>

当需要使用 CGLIB代理和@AspectJ自动代理支持，可以按照以下方式设置 <aop:aspectj-autoproxy>的proxy-target-class属性：

<aop:aspectj-autoproxy proxy-target-class="true"/>

而实际使用的过程中才会发现细节问题的差别，The devil is in the detail。

JDK动态代理：其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理。

CGLIB 代理：实现原理类似于 JDK 动态代理，只是它在运行期间生成的代理对象是针对目标类扩展的子类。CGLIB是高效的代码生成包，底层是依靠ASM（开源的Java字节码编辑类库）操作字节码实现

的，性能比JDK强。

expose-proxy：有时候目标对象内部的自我调用将无法实施切面中的增强，如下示例：

```
public interface AService {
    public void a();
    public void b();
}
@Service()
public class AServiceImpl1 implements AService{
    @Transactional(propagation = Propagation.REQUIRED)
public void a() {
      this.b();
   }
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void b() {
   }
}
```

此处的this指向目标对象，因此调用this.b()将不会执行b事务切面，即不会执行事务增强，因此b方法的事务定义“@Transactional(propagation = Propagation.REQUIRES_NEW)”将不会实施，为了解决这个问题，我们可以这样做：

```
<aop:aspectj-autoproxy expose-proxy="true"/>
```

然后将以上代码中的“this.b();”修改为“((AService) AopContext.currentProxy()).b();”即可。通过以上的修改便可以完成对a和b方法的同时增强。

最后注册组件并通知，便于监听器做进一步处理，这里就不再一一赘述了。

7.3 创建AOP代理

上文中讲解了通过自定义配置完成了对AnnotationAwareAspectJAutoProxyCreator 类型的自动注册，那么这个类到底做了什么工作来完成 AOP 的操作呢？首先我们看看 Annotation AwareAspectJAutoProxyCreator类的层次结构，如图7-1所示。

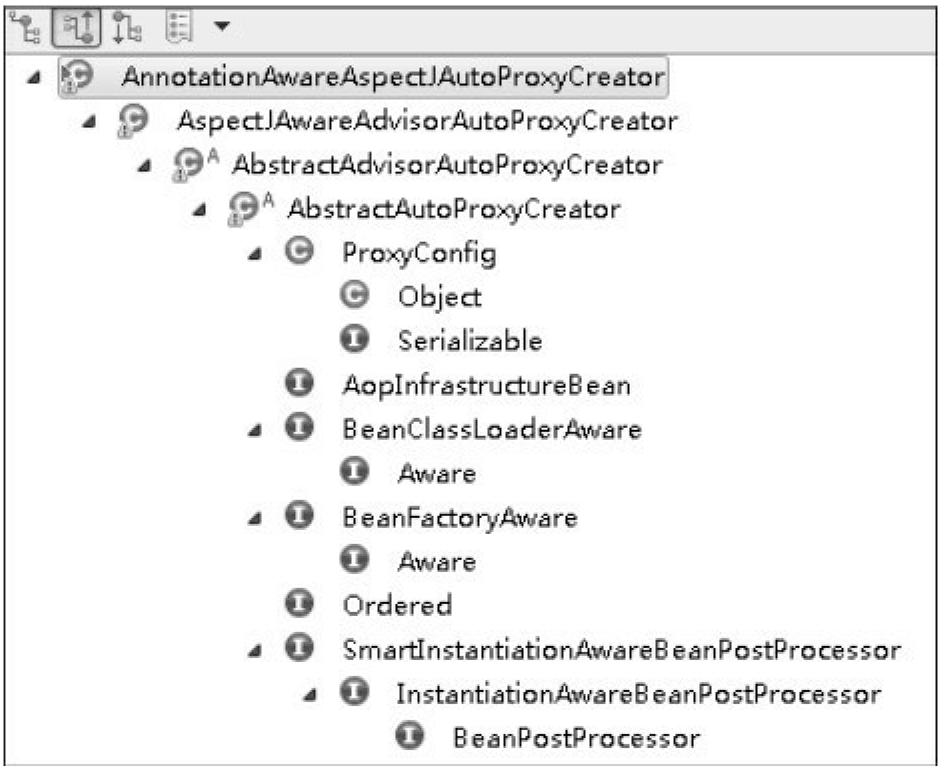


图7-1 AnnotationAwareAspectJAutoProxyCreator类的层次结构图

在类的层级中，我们看到 AnnotationAwareAspectJAutoProxyCreator实现了 BeanPostProcessor接口，而实现BeanPostProcessor后，当Spring加载这个Bean时会在实例化前调用其postProcess AfterInitialization方法，而我们对于AOP逻辑的分析也由此开始。

在父类AbstractAutoProxyCreator的postProcessAfterInitialization中代码如下：

```
public Object postProcessAfterInitialization(Object bean, String
beanName) throws BeansException {
    if (bean != null) {
      //根据给定的bean的class和name构建出个key，格式：
beanClassName_beanName
      Object cacheKey = getCacheKey(bean.getClass(), beanName);
      if (!this.earlyProxyReferences.contains(cacheKey)) {
      //如果它适合被代理,则需要封装指定bean。
        return wrapIfNecessary(bean, beanName, cacheKey);
      }
    }
    return bean;
}
protected Object wrapIfNecessary(Object bean, String beanName,
Object cacheKey) {
    //如果已经 处理过
    if (this.targetSourcedBeans.contains(beanName)) {
      return bean;
    }
    //无需增强
    if (this.nonAdvisedBeans.contains(cacheKey)) {
      return bean;
    }
  //给定的bean类是否代表一个基础设施类，基础设施类不应代理,或
者配置了指定bean不需要自动代理
    if (isInfrastructureClass(bean.getClass()) ||
shouldSkip(bean.getClass(),
```

```
beanName)) {
   this.nonAdvisedBeans.add(cacheKey);
    return bean;
}
//如果存在增强方法则创建代理
 Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(),
 beanName, null);
//如果获取到了增强则需要针对增强创建代理
 if (specificInterceptors != DO_NOT_PROXY) {
   this.advisedBeans.add(cacheKey);
   //创建代理
 Object proxy = createProxy(bean.getClass(), beanName,
specificInterceptors,
 new SingletonTargetSource(bean));
    this.proxyTypes.put(cacheKey, proxy.getClass());
    return proxy;
}
this.nonAdvisedBeans.add(cacheKey);
 return bean;
}
```

函数中我们已经看到了代理创建的雏形。当然，真正开始之前还需要经过一些判断，比如是否已经处理过或者是否是需要跳过的 bean，而真正创建代理的代码是从 getAdvicesAnd AdvisorsForBean开始的。

创建代理主要包含了两个步骤：

（1）获取增强方法或者增强器；

（2）根据获取的增强进行代理。

核心逻辑的时序图如图7-2所示。

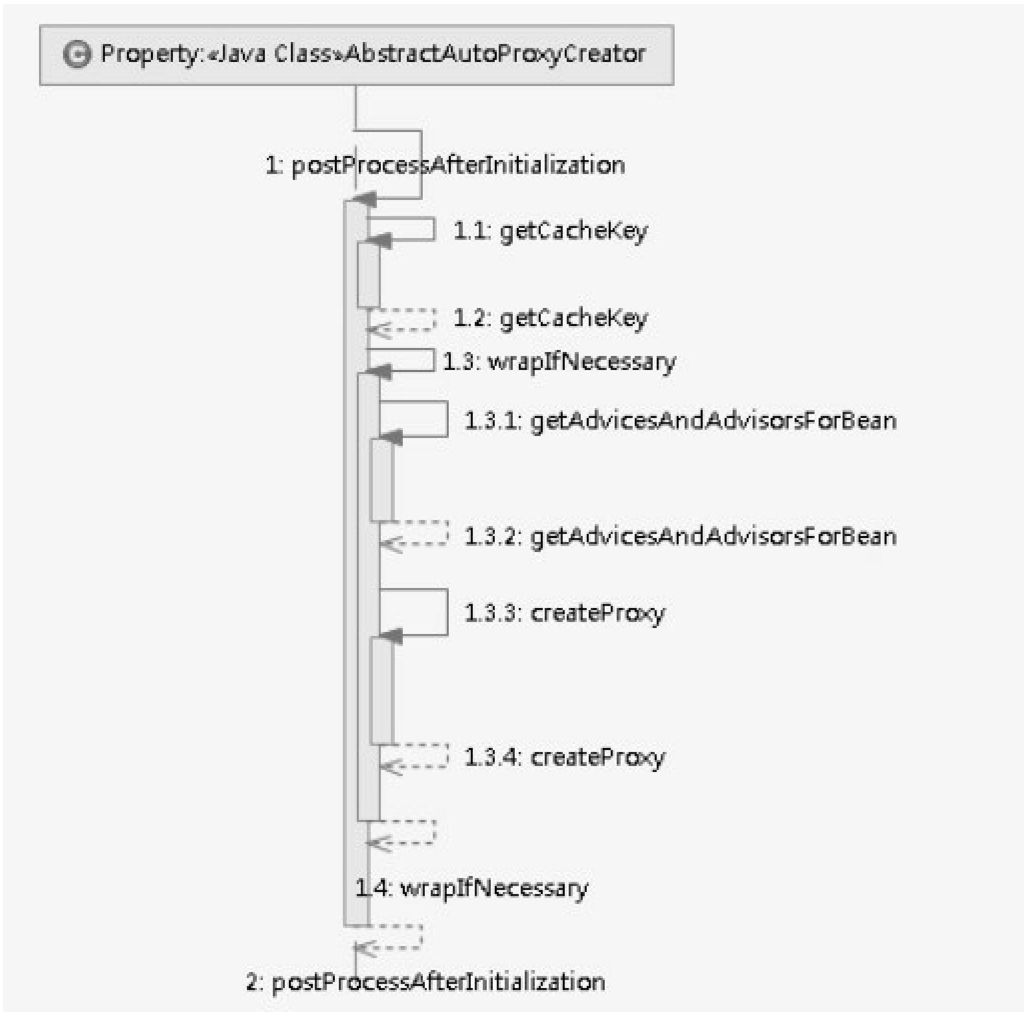


图7-2 AbstractAutoProxyCreator的postProcessAfterInitialization函数执行时序图

虽然看似简单，但是每个步骤中都经历了大量复杂的逻辑。首先来看看获取增强方法的实现逻辑。

```
@Override
protected Object[] getAdvicesAndAdvisorsForBean(Class beanClass, String beanName,
TargetSource targetSource) {
```

```
        List advisors = findEligibleAdvisors(beanClass, beanName);
        if (advisors.isEmpty()) {
            return DO_NOT_PROXY;
        }
        return advisors.toArray();
    }
    protected List<Advisor> findEligibleAdvisors(Class beanClass, String
beanName) {
        List<Advisor> candidateAdvisors = findCandidateAdvisors();
        List<Advisor> eligibleAdvisors =
findAdvisorsThatCanApply(candidateAdvisors,
        beanClass, beanName);
        extendAdvisors(eligibleAdvisors);
        if (!eligibleAdvisors.isEmpty()) {
            eligibleAdvisors = sortAdvisors(eligibleAdvisors);
        }
        return eligibleAdvisors;
    }
```

对于指定 bean 的增强方法的获取一定是包含两个步骤的，获取所有的增强以及寻找所有增强中适用于bean的增强并应用，那么findCandidateAdvisors与findAdvisorsThatCanApply便是做了这两件事情。当然，如果无法找到对应的增强器便返回 DO_NOT_PROXY，其中DO_NOT_PROXY=null。

7.3.1 获取增强器

由于我们分析的是使用注解进行的 AOP，所以对于findCandidateAdvisors 的实现其实是由

AnnotationAwareAspectJAutoProxyCreator 类完成的，我们继续跟踪AnnotationAwareAspectJAuto ProxyCreator的findCandidateAdvisors方法。

```
@Override
protected List<Advisor> findCandidateAdvisors() {
    //当使用注解方式配置AOP的时候并不是丢弃了对XML配置的支持，
    //在这里调用父类方法加载配置文件中的AOP声明
     List<Advisor> advisors = super.findCandidateAdvisors();
     // Build Advisors for all AspectJ aspects in the bean factory.
    advisors.addAll(this.aspectJAdvisorsBuilder.buildAspectJAdvisors());
     return advisors;
}
```

AnnotationAwareAspectJAutoProxyCreator间接继承了AbstractAdvisorAutoProxyCreator，在实现获取增强的方法中除了保留父类的获取配置文件中定义的增强外，同时添加了获取Bean的注解增强的功能，那么其实现正是由this.aspectJAdvisorsBuilder.buildAspectJAdvisors()来实现的。

在真正研究代码之前读者可以尝试着自己去想象一下解析思路，看看自己的实现与Spring是否有差别呢？或者我们一改以往的方式，先来了解函数提供的大概功能框架，读者可以在头脑中尝试实现这些功能点，看看是否有思路。

（1）获取所有beanName，这一步骤中所有在beanFacotry中注册的Bean都会被提取出来。

（2）遍历所有beanName，并找出声明AspectJ注解的类，进行进一步的处理。

（3）对标记为AspectJ注解的类进行增强器的提取。

（4）将提取结果加入缓存。

现在我们来看看函数实现，对Spring中所有的类进行分析，提取Advisor。

```
public List<Advisor> buildAspectJAdvisors() {
    List<String> aspectNames = null;
    synchronized (this) {
        aspectNames = this.aspectBeanNames;
        if (aspectNames == null) {
            List<Advisor> advisors = new LinkedList<Advisor>();
            aspectNames = new LinkedList<String>();
            //获取所有的beanName
            String[] beanNames
=BeanFactoryUtils.beanNamesForTypeIncludingAncestors
            (this.beanFactory, Object.class, true, false);
            //循环所有的beanName找出对应的增强方法
            for (String beanName : beanNames) {
                //不合法的bean则略过，由子类定义规则，默认返回true
                if (!isEligibleBean(beanName)) {
                continue;
                }
                //获取对应的bean的类型
                Class beanType = this.beanFactory.getType(beanName);
                if (beanType == null) {
                continue;
                }
                //如果存在Aspect注解
                if (this.advisorFactory.isAspect(beanType)) {
```

```
aspectNames.add(beanName);
AspectMetadata amd = new AspectMetadata(beanType,
beanName);
if (amd.getAjType().getPerClause().getKind() ==
PerClauseKind.
SINGLETON) {
    MetadataAwareAspectInstanceFactory factory =
    new BeanFactoryAspectInstanceFactory(this.
    beanFactory, beanName);
   //解析标记AspectJ注解中的增强方法
    List<Advisor> classAdvisors = this.advisorFactory.
   getAdvisors(factory);
    if (this.beanFactory.isSingleton(beanName)) {
        this.advisorsCache.put(beanName, classAdvisors);
   }
    else {
        this.aspectFactoryCache.put(beanName, factory);
   }
   advisors.addAll(classAdvisors);
}else {
    // Per target or per this.
    if (this.beanFactory.isSingleton(beanName)) {
        throw new IllegalArgumentException("Bean with
name
        '" + beanName +
        "' is a singleton, but aspect instantiation
        model is not singleton");
```

```
                    }
                    MetadataAwareAspectInstanceFactory factory =
                       new PrototypeAspectInstanceFactory(this.
                        beanFactory, beanName);
                    this.aspectFactoryCache.put(beanName, factory);

advisors.addAll(this.advisorFactory.getAdvisors(factory));
                }
            }
        }
        this.aspectBeanNames = aspectNames;
        return advisors;
    }
}
if (aspectNames.isEmpty()) {
    return Collections.EMPTY_LIST;
}
//记录在缓存中
List<Advisor> advisors = new LinkedList<Advisor>();
for (String aspectName : aspectNames) {
    List<Advisor> cachedAdvisors =
this.advisorsCache.get(aspectName);
    if (cachedAdvisors != null) {
       advisors.addAll(cachedAdvisors);
    }
    else {
        MetadataAwareAspectInstanceFactory factory =
```

```
this.aspectFactoryCache.
                get(aspectName);
                advisors.addAll(this.advisorFactory.getAdvisors(factory));
             }
          }
          return advisors;
       }
```

至此，我们已经完成了Advisor的提取，在上面的步骤中最为重要也最为繁杂的就是增强器的获取。而这一功能委托给了getAdvisors方法去实现（this.advisorFactory.getAdvisors(factory)）。

```
public List<Advisor>
getAdvisors(MetadataAwareAspectInstanceFactory maaif) {
          //获取标记为AspectJ的类
          final Class<?> aspectClass =
maaif.getAspectMetadata().getAspectClass();
          //获取标记为AspectJ的name
          final String aspectName =
maaif.getAspectMetadata().getAspectName();
          //验证
          validate(aspectClass);
          final MetadataAwareAspectInstanceFactory
lazySingletonAspectInstanceFactory
          =new LazySingletonAspectInstanceFactoryDecorator(maaif);
          final List<Advisor> advisors = new LinkedList<Advisor>();
          ReflectionUtils.doWithMethods(aspectClass, new
          ReflectionUtils.MethodCallback() {
             public void doWith(Method method) throws
```

```
IllegalArgumentException {
                    //声明为Pointcut的方法不处理
                    if (AnnotationUtils.getAnnotation(method, Pointcut.class) ==
null) {
                        Advisor advisor = getAdvisor(method,
lazySingletonAspectInstance
                        Factory, advisors.size(), aspectName);
                        if (advisor != null) {
                           advisors.add(advisor);
                        }
                    }
                }
            });
            if (!advisors.isEmpty() &&
lazySingletonAspectInstanceFactory.getAspect
            Metadata().isLazilyInstantiated()) {
               //如果寻找的增强器不为空而且又配置了增强延迟初始化那么
需要在首位加入同步实例化增强器
                Advisor instantiationAdvisor = new SyntheticInstantiationAdvisor
(lazySingleton
               AspectInstanceFactory);
                advisors.add(0, instantiationAdvisor);
            }
            //获取DeclareParents注解
            for (Field field : aspectClass.getDeclaredFields()) {
                Advisor advisor = getDeclareParentsAdvisor(field);
                if (advisor != null) {
```

```
            advisors.add(advisor);
        }
    }
    return advisors;
}
```

函数中首先完成了对增强器的获取，包括获取注解以及根据注解生成增强的步骤，然后考虑到在配置中可能会将增强配置成延迟初始化，那么需要在首位加入同步实例化增强器以保证增强使用之前的实例化，最后是对DeclareParents注解的获取，下面将详细介绍一下每个步骤。

1．普通增强器的获取

普通增强器的获取逻辑通过getAdvisor方法实现，实现步骤包括对切点的注解的获取以及根据注解信息生成增强。

```
public Advisor getAdvisor(Method candidateAdviceMethod,
MetadataAwareAspectInstanceFactory aif,
int declarationOrderInAspect, String aspectName) {
    validate(aif.getAspectMetadata().getAspectClass());
    //切点信息的获取
     AspectJExpressionPointcut ajexp =
        getPointcut(candidateAdviceMethod,
aif.getAspectMetadata().getAspectClass());
     if (ajexp == null) {
         return null;
    }
//根据切点信息生成增强器
     return new InstantiationModelAwarePointcutAdvisorImpl(
     this, ajexp, aif, candidateAdviceMethod, declarationOrderInAspect,
    aspectName);
```

}

（1）切点信息的获取。所谓获取切点信息就是指定注解的表达式信息的获取，如@Before("test()")。

```
private AspectJExpressionPointcut getPointcut(Method
candidateAdviceMethod, Class<?>
candidateAspectClass) {
    //获取方法上的注解
     AspectJAnnotation<?> aspectJAnnotation =
    AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod
    (candidateAdviceMethod);
     if (aspectJAnnotation == null) {
         return null;
    }
        //使用AspectJExpressionPointcut实例封装获取的信息
     AspectJExpressionPointcut ajexp =
     new AspectJExpressionPointcut(candidateAspectClass, new
String[0],
     new Class[0]);
    //提取得到的注解中的表达式如:
    //@Pointcut("execution(* *.*test*(..))")中的execution(* *.*test*(..))
ajexp.setExpression(aspectJAnnotation.getPointcutExpression());
     return ajexp;
}
protected static AspectJAnnotation
findAspectJAnnotationOnMethod(Method method) {
    //设置敏感的注解类
     Class<? extends Annotation>[] classesToLookFor = new Class[] {
```

```
        Before.class, Around.class, After.class, AfterReturning.class,
         AfterThrowing.class, Pointcut.class};
    for (Class<? extends Annotation> c : classesToLookFor) {
        AspectJAnnotation foundAnnotation = findAnnotation(method, c);
        if (foundAnnotation != null) {
            return foundAnnotation;
        }
    }
    return null;
}
//获取指定方法上的注解并使用AspectJAnnotation封装
private static <A extends Annotation> AspectJAnnotation<A>
findAnnotation(Method method,
Class<A> toLookFor) {
    A result = AnnotationUtils.findAnnotation(method, toLookFor);
    if (result != null) {
        return new AspectJAnnotation<A>(result);
    }
    else {
        return null;
    }
}
```

（2）根据切点信息生成增强。所有的增强都由 Advisor 的实现类 InstantiationModelAware PointcutAdvisorImpl统一封装的。

```
public
InstantiationModelAwarePointcutAdvisorImpl(AspectJAdvisorFactory af,
AspectJ
```

```
        ExpressionPointcut ajexp,
            MetadataAwareAspectInstanceFactory aif, Method method, int
declarationOrderInAspect,
            String aspectName) {
           //test()
            this.declaredPointcut = ajexp;
            //public void test.AspectJTest.beforeTest()
            this.method = method;
            this.atAspectJAdvisorFactory = af;
            this.aspectInstanceFactory = aif;
           //0
            this.declarationOrder = declarationOrderInAspect;
           //test.AspectJTest
            this.aspectName = aspectName;
            if (aif.getAspectMetadata().isLazilyInstantiated()) {
                // Static part of the pointcut is a lazy type.
                Pointcut preInstantiationPointcut =
                   Pointcuts.union(aif.getAspectMetadata().getPerClausePointcut(),
                   this.declaredPointcut);
                this.pointcut = new
PerTargetInstantiationModelPointcut(this.declaredPointcut,
                preInstantiationPointcut, aif);
                this.lazy = true;
            }else {
                // A singleton aspect.
                this.instantiatedAdvice = instantiateAdvice(this.declaredPointcut);
                this.pointcut = declaredPointcut;
```

```
            this.lazy = false;
        }
    }
```

在封装过程中只是简单地将信息封装在类的实例中，所有的信息单纯地赋值，在实例初始化的过程中还完成了对于增强器的初始化。因为不同的增强所体现的逻辑是不同的，比如@Before（“test()”）与@After（“test()”）标签的不同就是增强器增强的位置不同，所以就需要不同的增强器来完成不同的逻辑，而根据注解中的信息初始化对应的增强器就是在instantiateAdvice函数中实现的。

```
    private Advice instantiateAdvice(AspectJExpressionPointcut pcut) {
        return this.atAspectJAdvisorFactory.getAdvice(
        this.method, pcut, this.aspectInstanceFactory, this.declarationOrder,
       this.aspectName);
    }
    public Advice getAdvice(Method candidateAdviceMethod,
AspectJExpressionPointcut ajexp,
    MetadataAwareAspectInstanceFactory aif, int
declarationOrderInAspect,
    String aspectName) {
        Class<?> candidateAspectClass =
aif.getAspectMetadata().getAspectClass();
       validate(candidateAspectClass);
    AspectJAnnotation<?> aspectJAnnotation =
        AbstractAspectJAdvisorFactory.findAspectJAnnotationOnMethod
(candidate
       AdviceMethod);
    if (aspectJAnnotation == null) {
```

```
        return null;
    }
    // If we get here, we know we have an AspectJ method.
    // Check that it's an AspectJ-annotated class
    if (!isAspect(candidateAspectClass)) {
        throw new AopConfigException("Advice must be declared inside an
aspect type: " +
        "Offending method '" + candidateAdviceMethod + "' in class [" +
        candidateAspectClass.getName() + "]");
    }
    if (logger.isDebugEnabled()) {
        logger.debug("Found AspectJ method: " + candidateAdviceMethod);
    }
    AbstractAspectJAdvice SpringAdvice;
    //根据不同的注解类型封装不同的增强器
    switch (aspectJAnnotation.getAnnotationType()) {
      case AtBefore:
          SpringAdvice =
newAspectJMethodBeforeAdvice(candidateAdviceMethod, ajexp, aif);
          break;
      case AtAfter:
          SpringAdvice = newAspectJAfterAdvice(candidateAdviceMethod,
ajexp, aif);
          break;
      case AtAfterReturning:
          SpringAdvice =
newAspectJAfterReturningAdvice(candidateAdviceMethod,
```

```
                ajexp, aif);
            AfterReturning afterReturningAnnotation = (AfterReturning)
aspectJAnnotation.
           getAnnotation();
                if (StringUtils.hasText(afterReturningAnnotation.returning())) {

SpringAdvice.setReturningName(afterReturningAnnotation.returning());
               }
               break;
           case AtAfterThrowing:
                SpringAdvice = new
AspectJAfterThrowingAdvice(candidateAdviceMethod,
                ajexp, aif);
                AfterThrowing afterThrowingAnnotation = (AfterThrowing)
aspectJAnnotation.
               getAnnotation();
                if (StringUtils.hasText(afterThrowingAnnotation.throwing())) {

SpringAdvice.setThrowingName(afterThrowingAnnotation.throwing());
               }
               break;
           case AtAround:
                SpringAdvice = new
AspectJAroundAdvice(candidateAdviceMethod, ajexp, aif);
               break;
            case AtPointcut:
                if (logger.isDebugEnabled()) {
```

```
                logger.debug("Processing pointcut '" + candidateAdviceMethod.
                getName() + "'");
               }
                return null;
            default:
             throw new UnsupportedOperationException(
             "Unsupported advice type on method " +
candidateAdviceMethod);
        }
         // Now to configure the advice...
        SpringAdvice.setAspectName(aspectName);
        SpringAdvice.setDeclarationOrder(declarationOrderInAspect);
         String[] argNames =
this.parameterNameDiscoverer.getParameterNames (candidateAdvice
        Method);
         if (argNames != null) {
            SpringAdvice.setArgumentNamesFromStringArray(argNames);
        }
        SpringAdvice.calculateArgumentBindings();
         return SpringAdvice;
    }
```

从函数中可以看到，Spring 会根据不同的注解生成不同的增强器，例如 AtBefore 会对应AspectJMethodBeforeAdvice，而在AspectJMethodBeforeAdvice中完成了增强方法的逻辑。我们尝试分析下几个常用的增强器实现。

MethodBeforeAdviceInterceptor。

我们首先查看MethodBeforeAdviceInterceptor类的内部实现。

```
public class MethodBeforeAdviceInterceptor implements
MethodInterceptor, Serializable {
    private MethodBeforeAdvice advice;
    /**
     * Create a new MethodBeforeAdviceInterceptor for the given
advice.
     * @param advice the MethodBeforeAdvice to wrap
     */
    public MethodBeforeAdviceInterceptor(MethodBeforeAdvice
advice) {
        Assert.notNull(advice, "Advice must not be null");
        this.advice = advice;
    }
    public Object invoke(MethodInvocation mi) throws Throwable {
        this.advice.before(mi.getMethod(), mi.getArguments(),
mi.getThis() );
        return mi.proceed();
    }
}
```

其中的属性 MethodBeforeAdvice 代表着前置增强的 AspectJMethodBeforeAdvice，跟踪before方法：

```
public void before(Method method, Object[] args, Object target) throws
Throwable {
    invokeAdviceMethod(getJoinPointMatch(), null, null);
}
protected Object invokeAdviceMethod(JoinPointMatch jpMatch, Object
returnValue, Throwable
```

```
        ex) throws Throwable {
            return
invokeAdviceMethodWithGivenArgs(argBinding(getJoinPoint(), jpMatch,
            returnValue, ex));
        }
        protected Object invokeAdviceMethodWithGivenArgs(Object[] args)
throws Throwable {
            Object[] actualArgs = args;
            if (this.aspectJAdviceMethod.getParameterTypes().length == 0) {
                actualArgs = null;
            }
            try {
              ReflectionUtils.makeAccessible(this.aspectJAdviceMethod);
              //激活增强方法
                return
this.aspectJAdviceMethod.invoke(this.aspectInstanceFactory. getAspect
                Instance(), actualArgs);
            }
            catch (IllegalArgumentException ex) {
                throw new AopInvocationException("Mismatch on arguments to
advice method [" +
                    this.aspectJAdviceMethod + "]; pointcut expression [" +
                    this.pointcut.getPointcutExpression() + "]", ex);
            }
            catch (InvocationTargetException ex) {
                throw ex.getTargetException();
            }
```

}

invokeAdviceMethodWithGivenArgs方法中的aspectJAdviceMethod正是对于前置增强的方法，在这里实现了调用。

AspectJAfterAdvice。

后置增强与前置增强有稍许不一致的地方。回顾之前讲过的前置增强，大致的结构是在拦截器链中放置MethodBeforeAdviceInterceptor，而在MethodBeforeAdviceInterceptor中又放置了AspectJMethodBeforeAdvice，并在调用invoke时首先串联调用。但是在后置增强的时候却不一样，没有提供中间的类，而是直接在拦截器链中使用了中间的AspectJAfterAdvice。

```
public class AspectJAfterAdvice extends AbstractAspectJAdvice
implements MethodInterceptor,
    AfterAdvice {
      public AspectJAfterAdvice(
          Method aspectJBeforeAdviceMethod, AspectJExpressionPointcut
pointcut,
          AspectInstanceFactory aif) {
      super(aspectJBeforeAdviceMethod, pointcut, aif);
      }
      public Object invoke(MethodInvocation mi) throws Throwable {
          try {
              return mi.proceed();
          }
          finally {
            //激活增强方法
            invokeAdviceMethod(getJoinPointMatch(), null, null);
          }
```

```
    }
    public boolean isBeforeAdvice() {
        return false;
    }
    public boolean isAfterAdvice() {
        return true;
    }
}
```

2．增加同步实例化增强器

如果寻找的增强器不为空而且又配置了增强延迟初始化，那么就需要在首位加入同步实例化增强器。同步实例化增强器SyntheticInstantiationAdvisor如下：

```
protected static class SyntheticInstantiationAdvisor extends
DefaultPointcutAdvisor {
    public SyntheticInstantiationAdvisor(final
MetadataAwareAspectInstanceFactory aif) {
        super(aif.getAspectMetadata().getPerClausePointcut(), new
MethodBeforeAdvice() {
            //目标方法前调用，类似@Before
            public void before(Method method, Object[] args, Object target)
{
                //简单初始化aspect
                aif.getAspectInstance();
            }
        });
    }
}
```

3．获取**DeclareParents**注解

DeclareParents主要用于引介增强的注解形式的实现，而其实现方式与普通增强很类似，只不过使用DeclareParentsAdvisor对功能进行封装。

```
private Advisor getDeclareParentsAdvisor(Field introductionField) {
    DeclareParents declareParents = (DeclareParents)
introductionField.getAnnotation
   (DeclareParents.class);
    if (declareParents == null) {
        // Not an introduction field
        return null;
    }
    if (DeclareParents.class.equals(declareParents.defaultImpl())) {
        // This is what comes back if it wasn't set. This seems bizarre...
        // TODO this restriction possibly should be relaxed
        throw new IllegalStateException("defaultImpl must be set on
Declare
       Parents");
    }
    return new DeclareParentsAdvisor(
    introductionField.getType(), declareParents.value(), declareParents.
   defaultImpl());
}
```

7.3.2 寻找匹配的增强器

前面的函数中已经完成了所有增强器的解析，但是对于所有增强器来讲，并不一定都适用于当前的Bean，还要挑取出适合的增强器，也就

是满足我们配置的通配符的增强器。具体实现在findAdvisorsThatCanApply中。

```
protected List<Advisor> findAdvisorsThatCanApply(
List<Advisor> candidateAdvisors, Class beanClass, String beanName) {
    ProxyCreationContext.setCurrentProxiedBeanName(beanName);
    try {
        //过滤已经得到的advisors
        return AopUtils.findAdvisorsThatCanApply(candidateAdvisors,
beanClass);
    }
    finally {
        ProxyCreationContext.setCurrentProxiedBeanName(null);
    }
}
```

继续看findAdvisorsThatCanApply：

```
public static List<Advisor> findAdvisorsThatCanApply(List<Advisor>
candidateAdvisors,
Class<?> clazz) {
    if (candidateAdvisors.isEmpty()) {
        return candidateAdvisors;
    }
    List<Advisor> eligibleAdvisors = new LinkedList<Advisor>();
    //首先处理引介增强
    for (Advisor candidate : candidateAdvisors) {
        if (candidate instanceof IntroductionAdvisor &&
canApply(candidate, clazz)) {
            eligibleAdvisors.add(candidate);
```

```
            }
        }
        boolean hasIntroductions = !eligibleAdvisors.isEmpty();
        for (Advisor candidate : candidateAdvisors) {
            //引介增强已经处理
            if (candidate instanceof IntroductionAdvisor) {
                continue;
            }
            //对于普通bean的处理
            if (canApply(candidate, clazz, hasIntroductions)) {
                eligibleAdvisors.add(candidate);
            }
        }
        return eligibleAdvisors;
    }
```

findAdvisorsThatCanApply 函数的主要功能是寻找所有增强器中适用于当前 class 的增强器。引介增强与普通的增强是处理不一样的，所以分开处理。而对于真正的匹配在 canApply中实现。

```
public static boolean canApply(Advisor advisor, Class<?> targetClass, boolean
hasIntroductions) {
        if (advisor instanceof IntroductionAdvisor) {
            return ((IntroductionAdvisor) advisor).getClassFilter().matches
(targetClass);
        }else if (advisor instanceof PointcutAdvisor) {
            PointcutAdvisor pca = (PointcutAdvisor) advisor;
            return canApply(pca.getPointcut(), targetClass, hasIntroductions);
```

```
            }else {
                // It doesn't have a pointcut so we assume it applies.
                return true;
            }
        }
        public static boolean canApply(Pointcut pc, Class<?> targetClass,
boolean hasIntroductions) {
            Assert.notNull(pc, "Pointcut must not be null");
            if (!pc.getClassFilter().matches(targetClass)) {
                return false;
            }
            MethodMatcher methodMatcher = pc.getMethodMatcher();
            IntroductionAwareMethodMatcher
introductionAwareMethodMatcher = null;
            if (methodMatcher instanceof IntroductionAwareMethodMatcher) {
                introductionAwareMethodMatcher =
(IntroductionAwareMethodMatcher) methodMatcher;
            }
            Set<Class> classes = new HashSet<Class>
(ClassUtils.getAllInterfacesForClassAsSet
            (targetClass));
            classes.add(targetClass);
            //classes:[interface test.IITestBean, class test.TestBean]
            for (Class<?> clazz : classes) {
                Method[] methods = clazz.getMethods();
                for (Method method : methods) {
                    if ((introductionAwareMethodMatcher != null &&
```

introductionAwareMethodMatcher.matches(method,
targetClass,
hasIntroductions)) ||
methodMatcher.matches(method, targetClass)) {
return true;
}
}
}
return false;
}

7.3.3 创建代理

在获取了所有对应bean的增强器后，便可以进行代理的创建了。

```
protected Object createProxy(
Class<?> beanClass, String beanName, Object[] specificInterceptors,
TargetSource
targetSource) {
    ProxyFactory proxyFactory = new ProxyFactory();
    //获取当前类中相关属性
    proxyFactory.copyFrom(this);
    //决定对于给定的bean是否应该使用targetClass而不是他的接口代
理,
    //检查proxyTargeClass设置以及preserveTargetClass属性
    if (!shouldProxyTargetClass(beanClass, beanName)) {
        // Must allow for introductions; can't just set interfaces to
        // the target's interfaces only.
Class<?>[] targetInterfaces =
```

```
ClassUtils.getAllInterfacesForClass(beanClass, this.proxyClassLoader)
                for (Class<?> targetInterface : targetInterfaces) {
                   //添加代理接口
                   proxyFactory.addInterface(targetInterface);
                }
            }
            Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
            for (Advisor advisor : advisors) {
               //加入增强器
               proxyFactory.addAdvisor(advisor);
            }
            //设置要代理的类
            proxyFactory.setTargetSource(targetSource);
            //定制代理
            customizeProxyFactory(proxyFactory);
            //用来控制代理工厂被配置之后，是否还允许修改通知。
            //缺省值为false（即在代理被配置之后，不允许修改代理的配置）。
            proxyFactory.setFrozen(this.freezeProxy);
            if (advisorsPreFiltered()) {
               proxyFactory.setPreFiltered(true);
            }
            return proxyFactory.getProxy(this.proxyClassLoader);
        }
```

对于代理类的创建及处理，Spring委托给了ProxyFactory去处理，而在此函数中主要是对ProxyFactory的初始化操作，进而对真正的创建代理做准备，这些初始化操作包括如下内容。

（1）获取当前类中的属性。

（2）添加代理接口。

（3）封装Advisor并加入到ProxyFactory中。

（4）设置要代理的类。

（5）当然在Spring中还为子类提供了定制的函数customizeProxyFactory，子类可以在此函数中进行对ProxyFactory的进一步封装。

（6）进行获取代理操作。

其中，封装Advisor并加入到ProxyFactory中以及创建代理是两个相对繁琐的过程，可以通过ProxyFactory提供的addAdvisor方法直接将增强器置入代理创建工厂中，但是将拦截器封装为增强器还是需要一定的逻辑的。

```
protected Advisor[] buildAdvisors(String beanName, Object[] specificInterceptors) {
    //解析注册的所有interceptorName
    Advisor[] commonInterceptors = resolveInterceptorNames();
    List<Object> allInterceptors = new ArrayList<Object>();
    if (specificInterceptors != null) {
      //加入拦截器
      allInterceptors.addAll(Arrays.asList(specificInterceptors));
       if (commonInterceptors != null) {
          if (this.applyCommonInterceptorsFirst) {
             allInterceptors.addAll(0, Arrays.asList(commonInterceptors));
          }
          else {
             allInterceptors.addAll(Arrays.asList(commonInterceptors));
```

```
        }
      }
    }
    if (logger.isDebugEnabled()) {
      int nrOfCommonInterceptors = (commonInterceptors != null ?
commonInterceptors.
      length : 0);
    int nrOfSpecificInterceptors = (specificInterceptors != null ?
specificInterceptors.
    length : 0);
      logger.debug("Creating implicit proxy for bean '" + beanName + "'
with " +
      nrOfCommonInterceptors +
        " common interceptors and " + nrOfSpecificInterceptors + "
specific
       interceptors");
    }
    Advisor[] advisors = new Advisor[allInterceptors.size()];
    for (int i = 0; i < allInterceptors.size(); i++) {
      //拦截器进行封装转化为Advisor
      advisors[i] =
this.advisorAdapterRegistry.wrap(allInterceptors.get(i));
    }
    return advisors;
  }
  public Advisor wrap(Object adviceObject) throws
UnknownAdviceTypeException {
```

```
        //如果要封装的对象本身就是Advisor类型的那么无需再做过多处理
        if (adviceObject instanceof Advisor) {
            return (Advisor) adviceObject;
        }
        //因为此封装方法只对Advisor与Advice两种类型的数据有效，如果不是将不能封装
        if (!(adviceObject instanceof Advice)) {
            throw new UnknownAdviceTypeException(adviceObject);
        }
        Advice advice = (Advice) adviceObject;
        if (advice instanceof MethodInterceptor) {
            //如果是MethodInterceptor类型则使用DefaultPointcutAdvisor封装
            return new DefaultPointcutAdvisor(advice);
        }
        //如果存在Advisor的适配器那么也同样需要进行封装
        for (AdvisorAdapter adapter : this.adapters) {
            // Check that it is supported.
            if (adapter.supportsAdvice(advice)) {
                return new DefaultPointcutAdvisor(advice);
            }
        }
        throw new UnknownAdviceTypeException(advice);
    }
```

由于Spring中涉及过多的拦截器、增强器、增强方法等方式来对逻辑进行增强，所以非常有必要统一封装成Advisor来进行代理的创建，

完成了增强的封装过程，那么解析最重要的一步就是代理的创建与获取了。

```
public Object getProxy(ClassLoader classLoader) {
    return createAopProxy().getProxy(classLoader);
}
```

1. 创建代理

```
protected final synchronized AopProxy createAopProxy() {
     if (!this.active) {
        activate();
    }
    //创建代理
     return getAopProxyFactory().createAopProxy(this);
}
public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
    if (config.isOptimize() || config.isProxyTargetClass() ||
hasNoUserSuppliedProxy
     Interfaces(config)) {
        Class targetClass = config.getTargetClass();
        if (targetClass == null) {
            throw new AopConfigException("TargetSource cannot
determine target class: " +
                "Either an interface or a target is required for proxy
creation.");
        }
        if (targetClass.isInterface()) {
            return new JdkDynamicAopProxy(config);
```

```
            }
            if (!cglibAvailable) {
                throw new AopConfigException(
                    "Cannot proxy target class because CGLIB2 is not available.
" +
                    "Add CGLIB to the class path or specify proxy interfaces.");
            }
            return CglibProxyFactory.createCglibProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
```

到此已经完成了代理的创建，不管我们之前是否有阅读过Spring的源代码，但是都或多或少地听过对于Spring的代理中JDKProxy的实现和CglibProxy的实现。Spring是如何选取的呢？网上的介绍到处都是，现在我们就从源代码的角度分析，看看到底Spring是如何选择代理方式的。

从if中的判断条件可以看到3个方面影响着Spring的判断。

optimize：用来控制通过CGLIB创建的代理是否使用激进的优化策略。除非完全了解AOP代理如何处理优化，否则不推荐用户使用这个设置。目前这个属性仅用于CGLIB代理，对于JDK动态代理（缺省代理）无效。

proxyTargetClass：这个属性为true时，目标类本身被代理而不是目标类的接口。如果这个属性值被设为 true，CGLIB 代理将被创建，设置方式：<aop:aspectj-autoproxy proxy-target-class="true"/>。

hasNoUserSuppliedProxyInterfaces：是否存在代理接口。

下面是对JDK与Cglib方式的总结。

如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。

如果目标对象实现了接口，可以强制使用CGLIB实现AOP。

如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。

如何强制使用CGLIB实现AOP？

（1）添加CGLIB库，Spring_HOME/cglib/*.jar。

（2）在Spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>。

JDK动态代理和CGLIB字节码生成的区别？

JDK动态代理只能对实现了接口的类生成代理，而不能针对类。

CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，因为是继承，所以该类或方法最好不要声明成final。

2．获取代理

确定了使用哪种代理方式后便可以进行代理的创建了，但是创建之前有必要回顾一下两种方式的使用方法。

（1）JDK代理使用示例。

创建业务接口，业务对外提供的接口，包含着业务可以对外提供的功能。

```
public interface UserService {
    /**
      * 目标方法
      */
     public abstract void add();
}
```

创建业务接口实现类。

```
public class UserServiceImpl implements UserService {
    /* (non-Javadoc)
      * @see dynamic.proxy.UserService#add()
      */
    public void add() {
      System.out.println("--------------------add---------------");
    }
}
```

创建自定义的InvocationHandler，用于对接口提供的方法进行增强。

```
public class MyInvocationHandler implements InvocationHandler {
   // 目标对象
    private Object target;
   /**
      * 构造方法
      * @param target目标对象
      */
    public MyInvocationHandler(Object target) {
      super();
       this.target = target;
    }
   /**
      * 执行目标对象的方法
      */
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
      // 在目标对象的方法执行之前简单的打印一下
```

```
            System.out.println("------------------before------------------");
            // 执行目标对象的方法
             Object result = method.invoke(target, args);
            // 在目标对象的方法执行之后简单的打印一下
            System.out.println("-------------------after------------------");
             return result;
        }
        /**
            * 获取目标对象的代理对象
            * @return代理对象
            */
         public Object getProxy() {
             return
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
                 target.getClass().getInterfaces(), this);
        }
    }
```

最后进行测试，验证对于接口的增强是否起到作用。

```
    public class ProxyTest {
        @Test
         public void testProxy() throws Throwable {
            // 实例化目标对象
             UserService userService = new UserServiceImpl();
            // 实例化InvocationHandler
             MyInvocationHandler invocationHandler = new
MyInvocationHandler(userService);
            // 根据目标对象生成代理对象
```

UserService proxy = (UserService) invocationHandler.getProxy();

// 调用代理对象的方法

proxy.add();

}

}

执行结果如下：

-------------------before-------------

---------------------add--------------

--------------------after--------------

用起来很简单，其实这基本上就是 AOP 的一个简单实现了，在目标对象的方法执行之前和执行之后进行了增强。Spring的AOP实现其实也是用了Proxy和InvocationHandler这两个东西的。

我们再次来回顾一下使用 JDK 代理的方式，在整个创建过程中，对于 InvocationHandler的创建是最为核心的，在自定义的InvocationHandler中需要重写3个函数。

构造函数，将代理的对象传入。

invoke方法，此方法中实现了AOP增强的所有逻辑。

getProxy方法，此方法千篇一律，但是必不可少。

那么，我们看看Spring中的JDK代理实现是不是也是这么做的呢？继续之前的跟踪，到达JdkDynamicAopProxy的getProxy。

public Object getProxy(ClassLoader classLoader) {

if (logger.isDebugEnabled()) {

logger.debug("Creating JDK dynamic proxy: target source is " + this.

advised.getTargetSource());

}

Class[] proxiedInterfaces =

AopProxyUtils.completeProxiedInterfaces(this.advised);
findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}

通过之前的示例我们知道，JDKProxy 的使用关键是创建自定义的InvocationHandler，而InvocationHandler 中包含了需要覆盖的函数getProxy，而当前的方法正是完成了这个操作。再次确认一下JdkDynamicAopProxy也确实实现了InvocationHandler接口，那么我们就可以推断出，在JdkDynamicAopProxy中一定会有个invoke函数，并且JdkDynamicAopProxy会把AOP的核心逻辑写在其中。查看代码，果然有这样个函数：

```
public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
    MethodInvocation invocation;
    Object oldProxy = null;
    boolean setProxyContext = false;
    TargetSource targetSource = this.advised.targetSource;
    Class targetClass = null;
Object target = null;
try {
  //equals方法的处理
   if (!this.equalsDefined && AopUtils.isEqualsMethod(method)) {
      return equals(args[0]);
  }
  //hash方法的处理
   if (!this.hashCodeDefined &&
AopUtils.isHashCodeMethod(method)) {
```

```
            return hashCode();
        }
        /*
           * Class类的isAssignableFrom(Class cls)方法:
           * 如果调用这个方法的class或接口 与 参数cls表示的类或接口
相同，
           * 或者是参数cls表示的类或接口的父类，则返回true。
           * 形象地：自身类.class.isAssignableFrom(自身类或子类.class)
返回true
           * 例：
           *
System.out.println(ArrayList.class.isAssignableFrom(Object.class));
              //false
           *
System.out.println(Object.class.isAssignableFrom(ArrayList.class));
              //true
           */
      if (!this.advised.opaque && method.getDeclaringClass().isInterface()
&& method.getDeclaringClass().isAssignableFrom(Advised.class)) {
               return AopUtils.invokeJoinpointUsingReflection(this.advised,
method, args);
        }
         Object retVal;
        //有时候目标对象内部的自我调用将无法实施切面中的增强则需
要通过此属性暴露代理
         if (this.advised.exposeProxy) {
            oldProxy = AopContext.setCurrentProxy(proxy);
```

```
            setProxyContext = true;
        }
        target = targetSource.getTarget();
        if (target != null) {
            targetClass = target.getClass();
        }
        //获取当前方法的拦截器链
        List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice
        (method, targetClass);
        if (chain.isEmpty()) {
           //如果没有发现任何拦截器那么直接调用切点方法
            retVal = AopUtils.invokeJoinpointUsingReflection(target, method,
args);
        }else {
           //将拦截器封装在ReflectiveMethodInvocation，
           //以便于使用其proceed进行链接表用拦截器
           invocation = new ReflectiveMethodInvocation(proxy, target,
method,
            args, targetClass, chain);
           //执行拦截器链
           retVal = invocation.proceed();
        }
           //返回结果
            if (retVal != null && retVal == target &&
method.getReturnType(). IsInstance
            (proxy) &&
```

```
!RawTargetAccess.class.isAssignableFrom(method.getDeclaringClass())) {
                retVal = proxy;
               }
                return retVal;
            }
            finally {
                if (target != null && !targetSource.isStatic()) {
                    // Must have come from TargetSource.
                   targetSource.releaseTarget(target);
               }
                if (setProxyContext) {
                    // Restore old proxy.
                   AopContext.setCurrentProxy(oldProxy);
               }
            }
        }
```

上面的函数中最主要的工作就是创建了一个拦截器链，并使用ReflectiveMethodInvocation类进行了链的封装，而在ReflectiveMethodInvocation类的proceed方法中实现了拦截器的逐一调用，那么我们继续来探究，在proceed方法中是怎么实现前置增强在目标方法前调用后置增强在目标方法后调用的逻辑呢？

```
            if (this.currentInterceptorIndex ==
this.interceptorsAndDynamicMethodMatchers.
        public Object proceed() throws Throwable {
           // 执行完所有增强后执行切点方法
            size() - 1) {
```

```
            return invokeJoinpoint();
        }
        //获取下一个要执行的拦截器
        Object interceptorOrInterceptionAdvice =

this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIn
         if (interceptorOrInterceptionAdvice instanceof
InterceptorAndDynamicMethodMatcher) {
           //动态匹配,
            InterceptorAndDynamicMethodMatcher dm =
                (InterceptorAndDynamicMethodMatcher)
interceptorOrInterceptionAdvice;
            if (dm.methodMatcher.matches(this.method, this.targetClass,
this.arguments)) {
                return dm.interceptor.invoke(this);
            }else {
              //不匹配则不执行拦截器
                return proceed();
           }
         }else {
           /*普通拦截器，直接调用拦截器,比如：
              * ExposeInvocationInterceptor、
              * DelegatePerTargetObjectIntroductionInterceptor、
              * MethodBeforeAdviceInterceptor
              * AspectJAroundAdvice、
              * AspectJAfterAdvice
              */
```

```
//将this作为参数传递以保证当前实例中调用链的执行
    return ((MethodInterceptor)
interceptorOrInterceptionAdvice).invoke(this);
  }
}
```

在proceed方法中，或许代码逻辑并没有我们想象得那么复杂，ReflectiveMethodInvocation中的主要职责是维护了链接调用的计数器，记录着当前调用链接的位置，以便链可以有序地进行下去，那么在这个方法中并没有我们之前设想的维护各种增强的顺序，而是将此工作委托给了各个增强器，使各个增强器在内部进行逻辑实现。

（2）CGLIB使用示例。

CGLIB是一个强大的高性能的代码生成包。它广泛地被许多AOP的框架使用，例如Spring AOP和dynaop，为他们提供方法的Interception（拦截）。最流行的OR Mapping工具Hibernate也使用CGLIB来代理单端single-ended（多对一和一对一）关联（对集合的延迟抓取是采用其他机制实现的）。EasyMock和jMock是通过使用模仿（moke）对象来测试Java代码的包。它们都通过使用CGLIB来为那些没有接口的类创建模仿（moke）对象。

CGLIB包的底层通过使用一个小而快的字节码处理框架ASM，来转换字节码并生成新的类。除了CGLIB包，脚本语言例如Groovy和BeanShell，也是使用ASM来生成Java的字节码。当然不鼓励直接使用ASM，因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉。

我们先快速地了解CGLIB的使用示例。

```
import java.lang.reflect.Method;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
```

```
import net.sf.cglib.proxy.MethodProxy;
public class EnhancerDemo {
    public static void main(String[] args) {
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(EnhancerDemo.class);
        enhancer.setCallback(new MethodInterceptorImpl());
        EnhancerDemo demo = (EnhancerDemo) enhancer.create();
        demo.test();
        System.out.println(demo);
    }
    public void test() {
        System.out.println("EnhancerDemo test()");
    }
    private static class MethodInterceptorImpl implements
MethodInterceptor {
        @Override
        public Object intercept(Object obj, Method method, Object[] args,
        MethodProxy proxy) throws Throwable {
            System.err.println("Before invoke " + method);
            Object result = proxy.invokeSuper(obj, args);
            System.err.println("After invoke" + method);
            return result;
        }
    }
}
```

运行结果如下：

```
Before invoke public void EnhancerDemo.test()
```

EnhancerDemo test()

After invokepublic void EnhancerDemo.test()

Before invoke public java.lang.String java.lang.Object.toString()

Before invoke public native int java.lang.Object.hashCode()

After invokepublic native int java.lang.Object.hashCode()

After invokepublic java.lang.String java.lang.Object.toString()

EnhancerDemo$$EnhancerByCGLIB$$bc9b2066@1621e42

可以看到 System.out.println(demo), demo 首先调用了 toString()方法，然后又调用了hashCode，生成的对象为 EnhancerDemo$$EnhancerByCGLIB$$bc9b2066 的实例，这个类是运行时由CGLIB产生的。

完成CGLIB代理的类是委托给Cglib2AopProxy类去实现的，我们进入这个类一探究竟。

按照前面提供的示例，我们容易判断出来，Cglib2AopProxy的入口应该是在getProxy，也就是说在Cglib2AopProxy类的getProxy方法中实现了Enhancer的创建及接口封装。

```
public Object getProxy(ClassLoader classLoader) {
    if (logger.isDebugEnabled()) {
        logger.debug("Creating CGLIB2 proxy: target source is " +
this.advised.
       getTargetSource());
    }
    try {
        Class rootClass = this.advised.getTargetClass();
        Assert.state(rootClass != null, "Target class must be available for
creating
        a CGLIB proxy");
```

```
 Class proxySuperClass = rootClass;
 if (ClassUtils.isCglibProxyClass(rootClass)) {
     proxySuperClass = rootClass.getSuperclass();
     Class[] additionalInterfaces = rootClass.getInterfaces();
     for (Class additionalInterface : additionalInterfaces) {
        this.advised.addInterface(additionalInterface);
    }
}
//验证Class
validateClassIfNecessary(proxySuperClass);
//创建及配置Enhancer
 Enhancer enhancer = createEnhancer();
 if (classLoader != null) {
    enhancer.setClassLoader(classLoader);
     if (classLoader instanceof SmartClassLoader &&
     ((SmartClassLoader) classLoader).isClassReloadable(proxy
     SuperClass)) {
        enhancer.setUseCache(false);
    }
}
enhancer.setSuperclass(proxySuperClass);
 enhancer.setStrategy(new
UndeclaredThrowableStrategy(UndeclaredThrowable
Exception.class));

enhancer.setInterfaces(AopProxyUtils.completeProxiedInterfaces(this.advised
enhancer.setInterceptDuringConstruction(false);
```

```
//设置拦截器
 Callback[] callbacks = getCallbacks(rootClass);
enhancer.setCallbacks(callbacks);
 enhancer.setCallbackFilter(new ProxyCallbackFilter(
 this.advised.getConfigurationOnlyCopy(),
this.fixedInterceptorMap,
this.fixedInterceptorOffset));
 Class[] types = new Class[callbacks.length];
 for (int x = 0; x < types.length; x++) {
     types[x] = callbacks[x].getClass();
}
enhancer.setCallbackTypes(types);
//生成代理类以及创建代理
 Object proxy;
 if (this.constructorArgs != null) {
     proxy = enhancer.create(this.constructorArgTypes,
this.constructorArgs);
}
 else {
     proxy = enhancer.create();
}
 return proxy;
}
 catch (CodeGenerationException ex) {
 throw new AopConfigException("Could not generate CGLIB
subclass of class [" +
 this.advised.getTargetClass() + "]: " +
```

```
                "Common causes of this problem include using a final class or a
                non-visible class",
               ex);
          }
          catch (IllegalArgumentException ex) {
                throw new AopConfigException("Could not generate CGLIB
subclass of class [" +
                this.advised.getTargetClass() + "]: " +
                "Common causes of this problem include using a final class or a
                non-visible class",
               ex);
          }
          catch (Exception ex) {
                // TargetSource.getTarget() failed
                throw new AopConfigException("Unexpected AOP exception",
ex);
          }
     }
```

以上函数完整地阐述了一个创建Spring中的Enhancer的过程，读者可以参考Enhancer的文档查看每个步骤的含义，这里最重要的是通过getCallbacks方法设置拦截器链。

```
     private Callback[] getCallbacks(Class rootClass) throws Exception {
         //对于expose-proxy属性的处理
          boolean exposeProxy = this.advised.isExposeProxy();
          boolean isFrozen = this.advised.isFrozen();
          boolean isStatic = this.advised.getTargetSource().isStatic();
         //将拦截器封装在DynamicAdvisedInterceptor中
```

```
            Callback aopInterceptor = new
DynamicAdvisedInterceptor(this.advised);
            // Choose a "straight to target" interceptor. (used for calls that are
            // unadvised but can return this). May be required to expose the
proxy.
            Callback targetInterceptor;
            if (exposeProxy) {
                targetInterceptor = isStatic ?
                new StaticUnadvisedExposedInterceptor
(this.advised.getTargetSource().
                getTarget()) :
                new
DynamicUnadvisedExposedInterceptor(this.advised.getTargetSource());
            }else {
                targetInterceptor = isStatic ?
                new StaticUnadvisedInterceptor(this.advised.getTargetSource().
getTarget()) :
                new
DynamicUnadvisedInterceptor(this.advised.getTargetSource());
            }
            // Choose a "direct to target" dispatcher (used for
            // unadvised calls to static targets that cannot return this).
            Callback targetDispatcher = isStatic ?
                new StaticDispatcher(this.advised.getTargetSource().getTarget()) :
                new SerializableNoOp();
            Callback[] mainCallbacks = new Callback[]{
              //将拦截器链加入Callback中
```

```
            aopInterceptor,
             targetInterceptor, // invoke target without considering advice, if
optimized
             new SerializableNoOp(), // no override for methods mapped to this
             targetDispatcher, this.advisedDispatcher,
             new EqualsInterceptor(this.advised),
             new HashCodeInterceptor(this.advised)
        };
         Callback[] callbacks;
         // If the target is a static one and the advice chain is frozen,
         // then we can make some optimisations by sending the AOP calls
         // direct to the target using the fixed chain for that method.
         if (isStatic && isFrozen) {
             Method[] methods = rootClass.getMethods();
             Callback[] fixedCallbacks = new Callback[methods.length];
             this.fixedInterceptorMap = new HashMap<String, Integer>
(methods.length);
             // TODO: small memory optimisation here (can skip creation for
             // methods with no advice)
             for (int x = 0; x < methods.length; x++) {
                 List<Object> chain = this.advised.getInterceptorsAndDynamic
Interception
                 Advice(methods[x], rootClass);
                 fixedCallbacks[x] = new FixedChainStaticTargetInterceptor(
                 chain, this.advised.getTargetSource().getTarget(), this.advised.
                getTargetClass());
                 this.fixedInterceptorMap.put(methods[x].toString(), x);
```

```
            }
             // Now copy both the callbacks from mainCallbacks
             // and fixedCallbacks into the callbacks array.
             callbacks = new Callback[mainCallbacks.length +
fixedCallbacks.length];
             System.arraycopy(mainCallbacks, 0, callbacks, 0,
mainCallbacks.length);
             System.arraycopy(fixedCallbacks, 0, callbacks,
mainCallbacks.length,
            fixedCallbacks.length);
             this.fixedInterceptorOffset = mainCallbacks.length;
         }
         else {
             callbacks = mainCallbacks;
         }
         return callbacks;
     }
```

在getCallback中Spring考虑了很多情况，但是对于我们来说，只需要理解最常用的就可以了，比如将advised属性封装在DynamicAdvisedInterceptor并加入在callbacks中，这么做的目的是什么呢，如何调用呢？在前面的示例中，我们了解到 CGLIB 中对于方法的拦截是通过将自定义的拦截器（实现MethodInterceptor接口）加入Callback中并在调用代理时直接激活拦截器中的 intercept 方法来实现的，那么在 getCallback 中正是实现了这样一个目的，Dynamic AdvisedInterceptor继承自MethodInterceptor，加入Callback中后，在再次调用代理时会直接调用DynamicAdvisedInterceptor中的intercept方法，由此推断，对于CGLIB方式实现的代理，其核心逻辑必然在

DynamicAdvisedInterceptor中的intercept中。

```
    public Object intercept(Object proxy, Method method, Object[] args,
MethodProxy methodProxy)
    throws Throwable {
        Object oldProxy = null;
        boolean setProxyContext = false;
        Class targetClass = null;
        Object target = null;
        try {
            if (this.advised.exposeProxy) {
                // Make invocation available if necessary.
                oldProxy = AopContext.setCurrentProxy(proxy);
                setProxyContext = true;
            }
            target = getTarget();
            if (target != null) {
                targetClass = target.getClass();
            }
            //获取拦截器链
            List<Object> chain =
this.advised.getInterceptorsAndDynamicInterceptionAdvice
            (method, targetClass);
            Object retVal;
            if (chain.isEmpty() && Modifier.isPublic(method.getModifiers()))
{
                //如果拦截器链为空则直接激活原方法
                retVal = methodProxy.invoke(target, args);
```

```
            }else {
              //进入链
               retVal = new CglibMethodInvocation(proxy, target, method,
args,
               targetClass, chain, methodProxy).proceed();
            }
            retVal = massageReturnTypeIfNecessary(proxy, target, method,
retVal);
            return retVal;
        }
        finally {
            if (target != null) {
              releaseTarget(target);
            }
            if (setProxyContext) {
               // Restore old proxy.
              AopContext.setCurrentProxy(oldProxy);
            }
        }
    }
```

上述的实现与JDK方式实现代理中的invoke方法大同小异，都是首先构造链，然后封装此链进行串联调用，稍有些区别就是在JDK中直接构造ReflectiveMethodInvocation，而在cglib中使用CglibMethodInvocation。CglibMethodInvocation 继承自ReflectiveMethodInvocation，但是proceed方法并没有重写。

7.4 静态AOP使用示例

加载时织入（Load-Time Weaving，LTW）指的是在虚拟机载入字节码文件时动态织入AspectJ切面。Spring框架的值添加为AspectJ LTW在动态织入过程中提供了更细粒度的控制。使用Java（5+）的代理能使用一个叫“Vanilla”的AspectJ LTW，这需要在启动JVM的时候将某个JVM参数设置为开。这种JVM范围的设置在一些情况下或许不错，但通常情况下显得有些粗颗粒。而用Spring的LTW能让你在per-ClassLoader的基础上打开LTW，这显然更加细粒度并且对“单JVM多应用”的环境更具意义（例如在一个典型应用服务器环境中）。另外，在某些环境下，这能让你使用LTW而不对应用服务器的启动脚本做任何改动，不然则需要添加-javaagent:path/to/aspectjweaver.jar或者（以下将会提及的）-javaagent:path/to/Spring-agent.jar。开发人员只需简单修改应用上下文的一个或几个文件就能使用LTW，而不需依靠那些管理者部署配置，比如启动脚本的系统管理员。

我们还是以之前的 AOP 示例为基础，如果想从动态代理的方式改成静态代理的方式需要做如下改动。

（1）Spring全局配置文件的修改，加入LWT开关。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.Springframework.org/schema/aop"
   xmlns:context="http://www.Springframework.org/schema/context"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
   http://www.Springframework.org/schema/beans/Spring-
   beans-3.0.xsd
   http://www.Springframework.org/schema/aop
    http://www.Springframework.org/schema/aop/Spring-aop -3.0.xsd
```

```
            http://www.Springframework.org/schema/context http:
           //www.Springframework.org/schema/context/Spring-context-
3.0.xsd
    ">
    <aop:aspectj-autoproxy />
    <bean id="test" class="test.TestBean"/>
    <bean class="test.AspectJTest"/>
      <context:load-time-weaver />
    </beans>
```

（2）加入aop.xml。在class目录下的META-INF（没有则自己建立）文件夹下建立aop.xml，内容如下：

```
    <!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
    <aspectj>
      <weaver>
        <!-- only weave classes in our application-specific packages -->
        <include within="test.*" />
      </weaver>
      <aspects>
        <!-- weave in just this aspect -->
        <aspect name="test.AspectJTest" />
      </aspects>
    </aspectj>
```

主要是告诉AspectJ需要对哪个包进行织入，并使用哪些增强器。

（3）加入启动参数。如果是在Eclipse中启动的话需要加上启动参数，如图7-3所示。

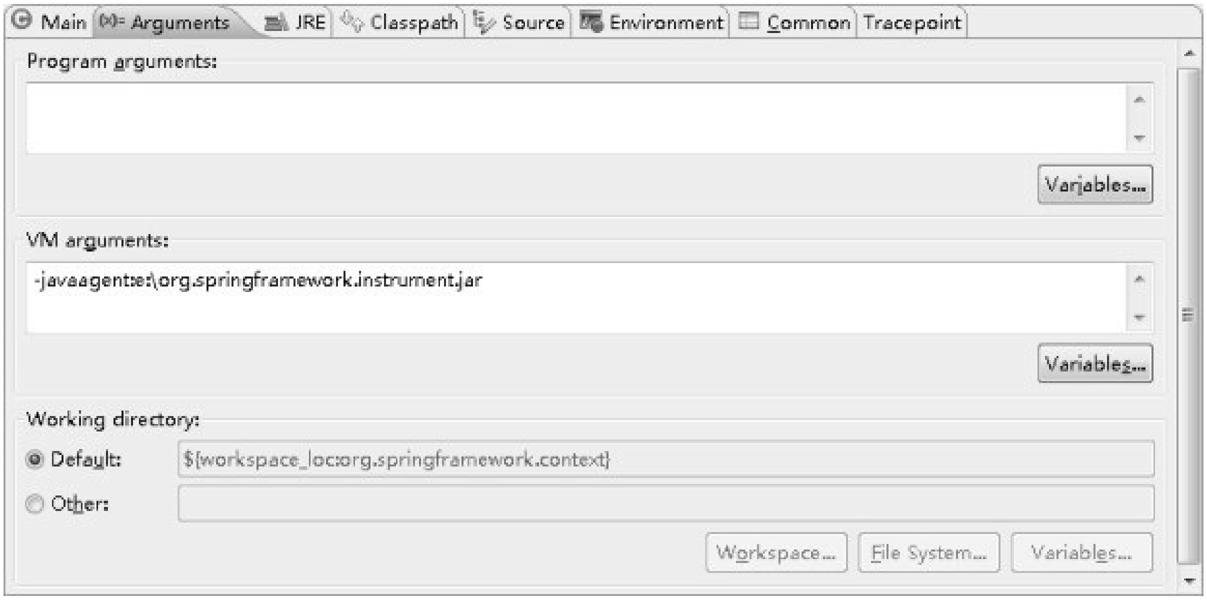


图7-3 eclipse使用AspectJ的配置

（4）测试。

```
public static void main(String[] args) {
    ApplicationContext bf = new
ClassPathXmlApplicationContext("aspectTest.xml");
    IITestBean bean=(IITestBean) bf.getBean("test");
   bean.testBeanM();
}
```

测试结果与动态AOP并无差别，打印出结果：

```
beforeTest
test
afterTest
```

7.5 创建AOP静态代理

AOP 的静态代理主要是在虚拟机启动时通过改变目标对象字节码的方式来完成对目标对象的增强，它与动态代理相比具有更高的效率，

因为在动态代理调用的过程中，还需要一个动态创建代理类并代理目标对象的步骤，而静态代理则是在启动时便完成了字节码增强，当系统再次调用目标类时与调用正常的类并无差别，所以在效率上会相对高些。

7.5.1 Instrumentation使用

Java在1.5引入 java.lang.instrument，你可以由此实现一个 Java agent，通过此 agent来修改类的字节码即改变一个类。本节会通过 Java Instrument实现一个简单的profiler。当然 instrument并不限于profiler，instrument它可以做很多事情，它类似一种更低级、更松耦合的AOP，可以从底层来改变一个类的行为。你可以由此产生无限的遐想。接下来要做的事情，就是计算一个方法所花的时间，通常我们会在代码中按以下方式编写。

在方法开头加入 long stime = System.nanoTime()；在方法结尾通过System.nanoTime()-stime得出方法所花时间。你不得不在想监控的每个方法中写入重复的代码，好一点的情况，你可以用AOP来干这事，但总是感觉有点别扭，这种profiler的代码还是要打包在你的项目中，Java Instrument使得这一切更干净。

（1）写ClassFileTransformer类。

```
package org.toy;
import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.CannotCompileException;
import javassist.ClassPool;
import javassist.CtBehavior;
import javassist.CtClass;
import javassist.NotFoundException;
```

```
import javassist.expr.ExprEditor;
import javassist.expr.MethodCall;
public class PerfMonXformer implements ClassFileTransformer {
public byte[] transform(ClassLoader loader, String className,
Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
byte[] classfileBuffer) throws IllegalClassFormatException {
    byte[] transformed = null;
    System.out.println("Transforming " + className);
    ClassPool pool = ClassPool.getDefault();
    CtClass cl = null;
    try {
        cl = pool.makeClass(new java.io.ByteArrayInputStream(
       classfileBuffer));
        if (cl.isInterface() == false) {
            CtBehavior[] methods = cl.getDeclaredBehaviors();
            for (int i = 0; i < methods.length; i++) {
                if (methods[i].isEmpty() == false) {
                  //修改method字节码
                      doMethod(methods[i]);
                  }
               }
                transformed = cl.toBytecode();
            }
        } catch (Exception e) {
            System.err.println("Could not instrument " + className
            + ", exception : " + e.getMessage());
        } finally {
```

```
                if (cl != null) {
                    cl.detach();
                }
            }
            return transformed;
        }
        private void doMethod(CtBehavior method) throws
NotFoundException,
        CannotCompileException {
            method.insertBefore("long stime = System.nanoTime();");
            method.insertAfter("System.out.println(/"leave
"+method.getName()+" and
            time:/"+(System.nanoTime()-stime));");
        }
    }
```

（2）编写agent类。

```
    package org.toy;
    import java.lang.instrument.Instrumentation;
    import java.lang.instrument.ClassFileTransformer;
    public class PerfMonAgent {
        static private Instrumentation inst = null;
        /**
            * This method is called before the application’s main-method is
called,
            * when this agent is specified to the Java VM.
            **/
        public static void premain(String agentArgs, Instrumentation _inst) {
```

```
            System.out.println("PerfMonAgent.premain() was called.");
            // Initialize the static variables we use to track information.
            inst = _inst;
            // Set up the class-file transformer.
            ClassFileTransformer trans = new PerfMonXformer();
            System.out.println("Adding a PerfMonXformer instance to the JVM.");
            inst.addTransformer(trans);
        }
    }
```

上面两个类就是agent的核心了，JVM启动时在应用加载前会调用PerfMonAgent.premain，然后 PerfMonAgent.premain 中实例化了一个定制的 ClassFileTransforme，即 PerfMonXformer并通过inst.addTransformer(trans)把PerfMonXformer的实例加入Instrumentation实例（由JVM传入），这就使得应用中的类加载时，PerfMonXformer.transform都会被调用，你在此方法中可以改变加载的类。真的有点神奇，为了改变类的字节码，我使用了JBoss的Javassist，虽然你不一定要这么用，但JBoss的Javassist真的很强大，能让你很容易地改变类的字节码。在上面的方法中我通过改变类的字节码，在每个类的方法入口中加入了 long stime = System.nanoTime()，在方法的出口加入了：

```
System.out.println("methodClassName.methodName:"+(System.nanoTime()-stime));
```

（3）打包agent。

对于agent的打包，有点讲究。

JAR的META-INF/MANIFEST.MF加入Premain-Class: xx，xx在此语境中就是我们的agent类，即org.toy.PerfMonAgent。

如果你的 agent类引入别的包，需使用Boot-Class-Path: xx，xx在此语境中就是上面提到的 JBoss javassit，即/home/pwlazy/.m2/repository/javassist/javassist/3.8.0 .GA/ javassist-3.8.0.GA.jar。

下面附上Maven的POM。

[xhtml]view plaincopyprint?

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-
  v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.toy</groupId>
  <artifactId>toy-inst</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>toy-inst</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>javassist</groupId>
      <artifactId>javassist</artifactId>
      <version>3.8.0.GA</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
```

```
                <artifactId>junit</artifactId>
                <version>3.8.1</version>
                <scope>test</scope>
            </dependency>
        </dependencies>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-jar-plugin</artifactId>
                    <version>2.2</version>
                    <configuration>
                        <archive>
                            <manifestEntries>
                                <Premain-Class>org.toy.PerfMonAgent</Premain-
Class>
                                <Boot-Class-
Path>/home/pwlazy/.m2/repository/javassist/javassist/3.8.0.GA/
                                javassist-3.8.0.GA.jar</Boot-Class-Path>
                            </manifestEntries>
                        </archive>
                    </configuration>
                </plugin>
                <plugin>
                    <artifactId>maven-compiler-plugin </artifactId >
                    <configuration>
                        <source> 1.6 </source >
```

```
            <target> 1.6 </target>
          </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

（4）打包应用。

```
package org.toy;
public class App {
    public static void main(String[] args) {
        new App().test();
    }
    public void test() {
        System.out.println("Hello World!!");
    }
}
```

Java选项中有-javaagent:xx，其中xx就是你的 agent JAR，Java通过此选项加载 agent，由agent来监控classpath下的应用。

最后的执行结果：

```
PerfMonAgent.premain() was called.
Adding a PerfMonXformer instance to the JVM.
Transforming org/toy/App
Hello World!!
java.io.PrintStream.println:314216
org.toy.App.test:540082
Transforming java/lang/Shutdown
Transforming java/lang/Shutdown$Lock
```

java.lang.Shutdown.runHooks:29124

java.lang.Shutdown.sequence:132768

由执行结果可以看出，执行顺序以及通过改变 org.toy.App 的字节码加入监控代码确实生效了。你也可以发现，通过Instrment实现agent使得监控代码和应用代码完全隔离了。

通过之前的两个小示例我们似乎已经有所体会，在 Spring 中的静态AOP 直接使用了AspectJ提供的方法，而AspectJ又是在Instrument基础上进行的封装。就以上面的例子来看，至少在AspectJ中会有如下功能。

（1）读取META-INF/aop.xml。

（2）将aop.xml中定义的增强器通过自定义的ClassFileTransformer织入对应的类中。

当然这都是AspectJ所做的事情，并不在我们讨论的范畴，Spring是直接使用AspectJ，也就是将动态代理的任务直接委托给了AspectJ，那么，Spring 怎么嵌入AspectJ的呢?同样我们还是从配置文件入手。

7.5.2 自定义标签

在Spring中如果需要使用AspectJ的功能，首先要做的第一步就是在配置文件中加入配置：<context:load-time-weaver/>。我们根据之前介绍的自定义命名空间的知识便可以推断，引用AspectJ的入口便是这里，可以通过查找load-time-weaver来找到对应的自定义命名处理类。

通过Eclipse提供的字符串搜索功能，我们找到了ContextNamespaceHandler，在其中有这样一段函数。

```
public void init() {
    registerBeanDefinitionParser("property-placeholder", new
PropertyPlaceholder
    BeanDefinitionParser());
    registerBeanDefinitionParser("property-override", new
```

PropertyOverrideBean

DefinitionParser());

registerBeanDefinitionParser("annotation-config", new AnnotationConfigBean

DefinitionParser());

registerBeanDefinitionParser("component-scan", new ComponentScanBeanDefinition

Parser());

registerBeanDefinitionParser("load-time-weaver", new LoadTimeWeaverBeanDefinition

Parser());

registerBeanDefinitionParser("Spring-configured", new SpringConfiguredBeanDefinition

Parser());

registerBeanDefinitionParser("mbean-export", new MBeanExportBeanDefinitionParser());

registerBeanDefinitionParser("mbean-server", new MBeanServerBeanDefinitionParser());

}

继续跟进LoadTimeWeaverBeanDefinitionParser，作为BeanDefinitionParser接口的实现类，他的核心逻辑是从parse函数开始的，而经过父类的封装，LoadTimeWeaverBeanDefinitionParser类的核心实现被转移到了doParse函数中，如下：

protected void doParse(Element element, ParserContext parserContext, BeanDefinition

Builder builder) {

builder.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);

```
            if
(isAspectJWeavingEnabled(element.getAttribute(ASPECTJ_WEAVING_AT
            parserContext)) {
                RootBeanDefinition weavingEnablerDef = new
RootBeanDefinition();
                    // ASPECTJ_WEAVING_ENABLER_CLASS_NAME =
                    //
"org.Springframework.context.weaving.AspectJWeavingEnabler";

weavingEnablerDef.setBeanClassName(ASPECTJ_WEAVING_ENABLER_

parserContext.getReaderContext().registerWithGeneratedName(weavingEnab
                if
(isBeanConfigurerAspectEnabled(parserContext.getReaderContext().getBean
                ClassLoader())) {
                    new SpringConfiguredBeanDefinitionParser().parse(element,
parserContext);
                }
            }
        }
```

其实之前在分析动态AOP也就是在分析配置<aop:aspectj-autoproxy />中已经提到了自定义配置的解析流程，对于<aop:aspectj-autoproxy/>的解析无非是以标签作为标志，进而进行相关处理类的注册，那么对于自定义标签<context:load-time-weaver />其实是起到了同样的作用。

上面函数的核心作用其实就是注册一个对于ApectJ处理的类org.Springframework.context. weaving.AspectJWeavingEnabler，它的注册流程总结起来如下。

（1）是否开启AspectJ。

之前虽然反复提到了在配置文件中加入了<context:load-time-weaver/>便相当于加入了AspectJ开关。但是，并不是配置了这个标签就意味着开启了AspectJ功能，这个标签中还有一个属性aspectj-weaving，这个属性有3个备选值，on、off和autodetect，默认为autodetect，也就是说，如果我们只是使用了<context:load-time-weaver/>，那么Spring会帮助我们检测是否可以使用AspectJ功能，而检测的依据便是文件META-INF/aop.xml是否存在，看看在Spring中的实现方式。

```
protected boolean isAspectJWeavingEnabled(String value,
ParserContext parserContext) {
        if ("on".equals(value)) {
            return true;
        }
        else if ("off".equals(value)) {
            return false;
        }
        else {
           //自动检测
            ClassLoader cl =
parserContext.getReaderContext().getResourceLoader().
           getClassLoader();
            return
(cl.getResource(AspectJWeavingEnabler.ASPECTJ_AOP_XML_RESOURC
!= null);
        }
    }
```

（2）将

org.Springframework.context.weaving.AspectJWeavingEnabler 封装在BeanDefinition中注册。

当通过AspectJ功能验证后便可以进行AspectJWeavingEnabler的注册了，注册的方式很简单，无非是将类路径注册在新初始化的RootBeanDefinition中，在RootBeanDefinition的获取时会转换成对应的class。

尽管在 init 方法中注册了AspectJWeavingEnabler，但是对于标签本身 Spring也会以 bean的形式保存，也就是当Spring解析到<context:load-time-weaver/>标签的时候也会产生一个bean，而这个bean中的信息是什么呢？

在LoadTimeWeaverBeanDefinitionParser类中有这样的函数：

```
@Override
protected String getBeanClassName(Element element) {
    if (element.hasAttribute(WEAVER_CLASS_ATTRIBUTE)) {
        return element.getAttribute(WEAVER_CLASS_ATTRIBUTE);
    }
    return DEFAULT_LOAD_TIME_WEAVER_CLASS_NAME;
}
    @Override
protected String resolveId(Element element, AbstractBeanDefinition definition,
ParserContext parserContext) {
    return ConfigurableApplicationContext.LOAD_TIME_WEAVER_BEAN_NAME;
}
```

其中，可以看到：

WEAVER_CLASS_ATTRIBUTE="weaver-class"

DEFAULT_LOAD_TIME_WEAVER_CLASS_NAME =

"org.Springframework.context.weaving.DefaultContextLoadTimeWeave

ConfigurableApplicationContext.LOAD_TIME_WEAVER_BEAN_NA

单凭以上的信息我们至少可以推断，当Spring在读取到自定义标签<context:load-time-weaver/>后会产生一个bean，而这个bean的id为loadTimeWeaver，class为org.Springframework.context.weaving.DefaultContextLoadTimeWeaver，也就是完成了DefaultContextLoadTimeWeaver类的注册。

完成了以上的注册功能后，并不意味这在Spring中就可以使用AspectJ了，因为我们还有一个很重要的步骤忽略了，就是LoadTimeWeaverAwareProcessor 的注册。在 AbstractApplicationContext 中的prepareBeanFactory函数中有这样一段代码：

```
//增加对AspectJ的支持
    if
(beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
        (beanFactory));
         beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor
         // Set a temporary ClassLoader for type matching.
         beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader (beanFactory.
        getBeanClassLoader()));
    }
```

在 AbstractApplicationContext 中的 prepareBeanFactory 函数是在容器初始化时候调用的，也就是说只有注册了LoadTimeWeaverAwareProcessor才会激活整个AspectJ的功能。

7.5.3 织入

当我们完成了所有的 AspectJ 的准备工作后便可以进行织入分析了，首先还是从LoadTimeWeaverAwareProcessor开始。

LoadTimeWeaverAwareProcessor实现BeanPostProcessor方法，那么对于BeanPostProcessor接口来讲，postProcessBeforeInitialization与postProcessAfterInitialization有着其特殊意义，也就是说在所有bean的初始化之前与之后都会分别调用对应的方法，那么在LoadTimeWeaverAwareProcessor中的postProcessBeforeInitialization函数中完成了什么样的逻辑呢？

```
public Object postProcessBeforeInitialization(Object bean, String beanName) throws
    BeansException {
        if (bean instanceof LoadTimeWeaverAware) {
            LoadTimeWeaver ltw = this.loadTimeWeaver;
            if (ltw == null) {
                Assert.state(this.beanFactory != null,
                "BeanFactory required if no LoadTimeWeaver explicitly
                specified");
                ltw = this.beanFactory.getBean(

ConfigurableApplicationContext.LOAD_TIME_WEAVER_BEAN_NAME,
                LoadTimeWeaver.class);
            }
            ((LoadTimeWeaverAware) bean).setLoadTimeWeaver(ltw);
        }
        return bean;
```

}

我们综合之前讲解的所有信息，将所有相关信息串联起来一起分析这个函数。

在LoadTimeWeaverAwareProcessor中的postProcessBeforeInitialization函数中，因为最开始的if判断注定这个后处理器只对LoadTimeWeaverAware类型的bean起作用，而纵观所有的bean，实现LoadTimeWeaver接口的类只有AspectJWeavingEnabler。

当在Spring中调用AspectJWeavingEnabler时，this.loadTimeWeaver尚未被初始化，那么，会直接调用beanFactory.getBean方法获取对应的DefaultContextLoadTimeWeaver类型的bean，并将其设置为AspectJWeavingEnabler类型bean的loadTimeWeaver属性中。当然AspectJWeavingEnabler同样实现了BeanClassLoaderAware以及Ordered接口，实现BeanClassLoaderAware接口保证了在bean初始化的时候调用AbstractAutowireCapableBeanFactory 的 invokeAwareMethods 的时候将beanClassLoader赋值给当前类。而实现Ordered接口则保证在实例化bean时当前bean会被最先初始化。

而 DefaultContextLoadTimeWeaver 类又同时实现了LoadTimeWeaver、BeanClassLoaderAware以及DisposableBean。其中DisposableBean接口保证在bean销毁时会调用destroy方法进行bean的清理，而 BeanClassLoaderAware 接口则保证在 bean 的初始化调用AbstractAutowireCapable BeanFactory的invokeAwareMethods时调用setBeanClassLoader方法。

```
public void setBeanClassLoader(ClassLoader classLoader) {
    LoadTimeWeaver serverSpecificLoadTimeWeaver =
createServerSpecificLoadTimeWeaver
   (classLoader);
    if (serverSpecificLoadTimeWeaver != null) {
```

```
                if (logger.isInfoEnabled()) {
                    logger.info("Determined server-specific load-time weaver: " +
                   serverSpecificLoadTimeWeaver.getClass().getName());
               }
                this.loadTimeWeaver = serverSpecificLoadTimeWeaver;
            }else if
(InstrumentationLoadTimeWeaver.isInstrumentationAvailable()) {
               //检查当前虚拟机中的Instrumentation实例是否可用
                logger.info("Found Spring's JVM agent for instrumentation");
               this.loadTimeWeaver = new
InstrumentationLoadTimeWeaver(classLoader);
            }else {
                try {
                    this.loadTimeWeaver = new
ReflectiveLoadTimeWeaver(classLoader);
                    logger.info("Using a reflective load-time weaver for class
loader: " +
                   this.loadTimeWeaver.getInstrumentableClassLoader().
                   getClass().getName());
               }
                catch (IllegalStateException ex) {
                    throw new IllegalStateException(ex.getMessage() + " Specify a
custom
                    LoadTimeWeaver or start your " +
                    "Java virtual machine with Spring's agent: -javaagent:org.
                   Springframework.instrument.jar");
               }
```

}

}

上面的函数中有一句很容易被忽略但是很关键的代码：

this.loadTimeWeaver = new InstrumentationLoadTimeWeaver(classLoader);

这句代码不仅仅是实例化了一个InstrumentationLoadTimeWeaver类型的实例，而且在实例化过程中还做了一些额外的操作。

在实例化的过程中会对当前的this.instrumentation属性进行初始化，而初始化的代码如下：this.instrumentation = getInstrumentation()，也就是说在 InstrumentationLoadTimeWeaver实例化后其属性Instrumentation已经被初始化为代表着当前虚拟机的实例了。综合我们讲过的例子，对于注册转换器，如addTransformer函数等，便可以直接使用此属性进行操作了。

也就是经过以上程序的处理后，在Spring中的bean之间的关系如下：

AspectJWeavingEnabler 类型的 bean 中的 loadTimeWeaver 属性被初始化为 Default ContextLoadTimeWeaver类型的bean；

DefaultContextLoadTimeWeaver 类型的 bean 中的 loadTimeWeaver 属性被初始化为InstrumentationLoadTimeWeaver。

因为AspectJWeavingEnabler类同样实现了BeanFactoryPostProcessor，所以当所有bean解析结束后会调用其postProcessBeanFactory方法。

```
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws
BeansException {
    enableAspectJWeaving(this.loadTimeWeaver, this.beanClassLoader);
}
```

```
public static void enableAspectJWeaving(LoadTimeWeaver
weaverToUse, ClassLoader
beanClassLoader) {
    if (weaverToUse == null) {
//此时已经被初始化为DefaultContextLoadTimeWeaver
    if (InstrumentationLoadTimeWeaver.isInstrumentationAvailable()) {
        weaverToUse = new
InstrumentationLoadTimeWeaver(beanClassLoader);
    }
    else {
        throw new IllegalStateException("No LoadTimeWeaver
available");
    }
}
//使用DefaultContextLoadTimeWeaver类型的bean中的
loadTimeWeaver属性注册转换器
    weaverToUse.addTransformer(new
AspectJClassBypassingClassFileTransformer(
    new ClassPreProcessorAgentAdapter()));
}
private static class AspectJClassBypassingClassFileTransformer
implements Class
FileTransformer {
    private final ClassFileTransformer delegate;
public
AspectJClassBypassingClassFileTransformer(ClassFileTransformer delegate)
{
```

```
        this.delegate = delegate;
    }
    public byte[] transform(ClassLoader loader, String className, Class<?>
classBeingRedefined,
    ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
IllegalClassFormatException {
        if (className.startsWith("org.aspectj") || className.startsWith
        ("org/aspectj")) {
            return classfileBuffer;
        }
        //委托给AspectJ代理继续处理
        return this.delegate.transform(loader, className,
classBeingRedefined,
        protectionDomain, classfileBuffer);
        }
    }
```

AspectJClassBypassingClassFileTransformer的作用仅仅是告诉AspectJ以org.aspectj开头的或者org/aspectj开头的类不进行处理。

第二部分　企业应用



第8章　数据库连接JDBC

JDBC（Java Data Base Connectivity，Java数据库连接）是一种用于执行SQL语句的 Java API，可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。JDBC为数据库开发人员提供了一个标准的API，据此可以构建更高级的工具和接口，使数据库开发人员能够用纯 Java API编写数据库应用程序，并且可跨平台运行，并且不受数据库供应商的限制。

JDBC连接数据库的流程及其原理如下。

（1）在开发环境中加载指定数据库的驱动程序。接下来的实验中，使用的数据库是MySQL，所以需要去下载 MySQL 支持 JDBC 的驱动程序（最新的版本是 mysql-connector-java-5.1.18-bin.jar），将下载得到的驱动程序加载进开发环境中（开发环境是 MyEclipse，具体示例时会讲解如何加载）。

（2）在Java程序中加载驱动程序。在Java程序中，可以通过“Class.forName("指定数据库的驱动程序")”的方式来加载添加到开发环境中的驱动程序，例如加载 MySQL 的数据驱动程序的代码为 Class.forName("com.mysql.jdbc.Driver")。

（3）创建数据连接对象。通过 DriverManager 类创建数据库连接对象 Connection。DriverManager类作用于Java程序和JDBC驱动程序之间，用于检查所加载的驱动程序是否可以建立连接，然后通过它的 getConnection方法根据数据库的URL、用户名和密码，创建一个JDBC Connection对象，例如：Connection connection = DriverManager.geiConnection("连接数据库的URL", "用户名", "密码”)。其中，URL=协议名+IP地址（域名）+端口+数据库名称；用户名和密码是指登录数据库时所使用的用户名和密码。具体示例创建MySQL的数据库连接代码如下：

Connection connectMySQL =
DriverManager.geiConnection(“jdbc:mysql://localhost:3306/
myuser","root" ,"root");

（4）创建Statement对象。Statement类的主要是用于执行静态SQL语句并返回它所生成结果的对象。通过Connection对象的createStatement()方法可以创建一个Statement对象。例如：Statement statament = connection.createStatement()。具体示例创建Statement对象代码如下：

Statement statamentMySQL =connectMySQL.createStatement();

（5）调用Statement对象的相关方法执行相对应的SQL语句。通过execuUpdate()方法来对数据更新，包括插入和删除等操作，例如向staff表中插入一条数据的代码：

statement.excuteUpdate("INSERT INTO staff(name, age, sex,address, depart,
worklen,wage)" + " VALUES ('Tom1', 321, 'M', 'china','Personnel','3','3000') ") ;

通过调用Statement对象的executeQuery()方法进行数据的查询，而查询结果会得到ResulSet对象，ResulSet 表示执行查询数据库后返回的数据的集合，ResulSet对象具有可以指向当前数据行的指针。通过该对象的 next()方法，使得指针指向下一行，然后将数据以列号或者字段名取出。如果当next()方法返回null，则表示下一行中没有数据存在。使用示例代码如下：

ResultSet resultSel = statement.executeQuery("select * from staff");

（6）关闭数据库连接。使用完数据库或者不需要访问数据库时，通过Connection的close()方法及时关闭数据连接。

8.1 Spring连接数据库程序实现（JDBC）

Spring中的JDBC连接与直接使用JDBC去连接还是有所差别的，Spring对JDBC做了大量封装，消除了冗余代码，使得开发量大大减小。下面通过一个小例子让大家简单认识Spring中的JDBC操作。

（1）创建数据表结构。

```
CREATE TABLE 'user' (
    'id' int(11) NOT NULL auto_increment,
    'name' varchar(255) default NULL,
    'age' int(11) default NULL,
    'sex' varchar(255) default NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

（2）创建对应数据表的PO。

```
public class User {
    private int id;
    private String name;
    private int age;
private String sex;
   //省略set/get方法
}
```

（3）创建表与实体间的映射。

```
public class UserRowMapper implements RowMapper {
   @Override
    public Object mapRow(ResultSet set, int index) throws
SQLException {
        User person = new User(set.getInt("id"), set.getString("name"), set
        .getInt("age"), set.getString("sex"));
        return person;
```

```
    }
}
```

（4）创建数据操作接口。

```
public interface UserService {
    public void save(User user);
    public List<User> getUsers();
}
```

（5）创建数据操作接口实现类。

```
public class UserServiceImpl implements UserService {
    private JdbcTemplate jdbcTemplate;
    // 设置数据源
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    public void save(User user) {
        jdbcTemplate.update("insert into user(name,age,sex)values(?,?,?)",
        new Object[] { user.getName(), user.getAge(),
        user.getSex() }, new int[] { java.sql.Types.VARCHAR,
        java.sql.Types.INTEGER, java.sql.Types.VARCHAR });
    }
    @SuppressWarnings("unchecked")
    public List<User> getUsers() {
        List<User> list = jdbcTemplate.query("select * from user", new
UserRowMapper());
        return list;
    }
}
```

（6）创建Spring配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
        http://www.Springframework.org/schema/beans/Spring-beans-
2.5.xsd
        ">
      <!--配置数据源 -->
      <bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
         <property name="driverClassName"
value="com.mysql.jdbc.Driver" />
         <property name="url"
           value="jdbc:mysql://localhost:3306/lexueba" />
         <property name="username" value="root" />
         <property name="password" value="haojia0421xixi" />
         <!--连接池启动时的初始值 -->
         <property name="initialSize" value="1" />
         <!--连接池的最大值 -->
         <property name="maxActive" value="300" />
        <!-- 最大空闲值.当经过一个高峰时间后，连接池可以慢慢将已
经用不到的连接慢慢释放一部分，一直减
        少到maxIdle为止 -->
         <property name="maxIdle" value="2" />
```

```
<!--最小空闲值.当空闲的连接数少于阀值时，连接池就会预申
请去一些连接，以免洪峰来时来不及申请 -->
<property name="minIdle" value="1" />
</bean>
<!--配置业务bean：PersonServiceBean -->
<bean id="userService" class="service.UserServiceImpl">
<!--向属性dataSource注入数据源 -->
<property name="dataSource" ref="dataSource"></property>
</bean>
</beans>
```

（7）测试。

```
public class SpringJDBCTest {
    public static void main(String[] args) {
        ApplicationContext act = new
ClassPathXmlApplicationContext("bean.xml");
        UserService userService = (UserService)
act.getBean("userService");
        User user = new User();
        user.setName("张三");
        user.setAge(20);
        user.setSex("男");
        // 保存一条记录
        userService.save(user);
        List<User> person1 = userService.getUsers();
        System.out.println("++++++++得到所有User");
        for (User person2 : person1) {
            System.out.println(person2.getId() + " " + person2.getName()
```

```
            + " " + person2.getAge() + " " + person2.getSex());
        }
    }
}
```

8.2 save/update功能的实现

我们以上面的例子为基础开始分析Spring中对JDBC的支持，首先寻找整个功能的切入点，在示例中我们可以看到所有的数据库操作都封装在了 UserServiceImpl 中，而 UserServiceImpl中的所有数据库操作又以其内部属性jdbcTemplate为基础。这个jdbcTemplate可以作为源码分析的切入点，我们一起看看它是如何实现又是如何被初始化的。

在 UserServiceImpl 中 jdbcTemplate 的初始化是从 setDataSource 函数开始的，DataSource实例通过参数注入，DataSource 的创建过程是引入第三方的连接池，这里不做过多介绍。DataSource是整个数据库操作的基础，里面封装了整个数据库的连接信息。我们首先以保存实体类为例进行代码跟踪。

```
public void save(User user) {
    jdbcTemplate.update("insert into user(name,age,sex)values(?,?,?)",
     new Object[] { user.getName(), user.getAge(),
     user.getSex() }, new int[] { java.sql.Types.VARCHAR,
     java.sql.Types.INTEGER, java.sql.Types.VARCHAR });
}
```

对于保存一个实体类来讲，在操作中我们只需要提供 SQL 语句以及语句中对应的参数和参数类型，其他操作便可以交由Spring来完成了，这些工作到底包括什么呢？进入jdbcTemplate中的update方法。

```
public int update(String sql, Object[] args, int[] argTypes) throws
```

```
DataAccessException {
        return update(sql, newArgTypePreparedStatementSetter(args,
argTypes));
    }
    public int update(String sql, PreparedStatementSetter pss) throws
DataAccessException {
        return update(new SimplePreparedStatementCreator(sql), pss);
    }
```

进入 update 方法后，Spring 并不是急于进入核心处理操作，而是先做足准备工作，使用ArgTypePreparedStatementSetter对参数与参数类型进行封装，同时又使用Simple PreparedStatement Creator对SQL语句进行封装。至于为什么这么封装，暂且留下悬念。

经过了数据封装后便可以进入了核心的数据处理代码了。

```
    protected int update(final PreparedStatementCreator psc, final
PreparedStatementSetter pss)
    throws DataAccessException {
        logger.debug("Executing prepared SQL update");
        return execute(psc, new PreparedStatementCallback<Integer>() {
          public Integer doInPreparedStatement(PreparedStatement ps)
throws
          SQLException {
            try {
              if (pss != null) {
                //设置PreparedStatement所需的全部参数。
                pss.setValues(ps);
              }
              int rows = ps.executeUpdate();
```

```
                    if (logger.isDebugEnabled()) {
                        logger.debug("SQL update affected " + rows + " rows");
                    }
                    return rows;
                }
                finally {
                    if (pss instanceof ParameterDisposer) {
                        ((ParameterDisposer) pss).cleanupParameters();
                    }
                }
            }
        });
    }
```

如果读者了解过其他操作方法，可以知道 execute 方法是最基础的操作，而其他操作比如update、query等方法则是传入不同的PreparedStatementCallback参数来执行不同的逻辑。

8.2.1 基础方法execute

execute作为数据库操作的核心入口，将大多数数据库操作相同的步骤统一封装，而将个性化的操作使用参数PreparedStatementCallback进行回调。

```
    public <T> T execute(PreparedStatementCreator psc, PreparedStatementCallback<T> action)
    throws DataAccessException {
        Assert.notNull(psc, "PreparedStatementCreator must not be null");
        Assert.notNull(action, "Callback object must not be null");
        if (logger.isDebugEnabled()) {
```

```
        String sql = getSql(psc);
        logger.debug("Executing prepared SQL statement" + (sql != null ?
" [" + sql + "]" :
       ""));
    }
    //获取数据库连接
    Connection con = DataSourceUtils.getConnection(getDataSource());
    PreparedStatement ps = null;
    try {
        Connection conToUse = con;
        if (this.nativeJdbcExtractor != null &&
       this.nativeJdbcExtractor.isNativeConnectionNecessaryForNative
        PreparedStatements()) {
            conToUse =
this.nativeJdbcExtractor.getNativeConnection(con);
       }
        ps = psc.createPreparedStatement(conToUse);
       //应用用户设定的输入参数
       applyStatementSettings(ps);
        PreparedStatement psToUse = ps;
        if (this.nativeJdbcExtractor != null) {
            psToUse =
this.nativeJdbcExtractor.getNativePreparedStatement(ps);
       }
       //调用回调函数
        T result = action.doInPreparedStatement(psToUse);
       handleWarnings(ps);
```

```
            return result;
        }
        catch (SQLException ex) {
            //释放数据库连接避免当 异常转换器没有被初始化的时候出现
潜在的连接池死锁
            if (psc instanceof ParameterDisposer) {
                ((ParameterDisposer) psc).cleanupParameters();
            }
            String sql = getSql(psc);
            psc = null;
            JdbcUtils.closeStatement(ps);
            ps = null;
            DataSourceUtils.releaseConnection(con, getDataSource());
            con = null;
            throw
getExceptionTranslator().translate("PreparedStatementCallback", sql, ex);
        }
        finally {
            if (psc instanceof ParameterDisposer) {
                ((ParameterDisposer) psc).cleanupParameters();
            }
            JdbcUtils.closeStatement(ps);
            DataSourceUtils.releaseConnection(con, getDataSource());
        }
    }
```

以上方法对常用操作进行了封装，包括如下几项内容。

1．获取数据库连接

获取数据库连接也并非直接使用 dataSource.getConnection()方法那么简单，同样也考虑了诸多情况。

```
public static Connection doGetConnection(DataSource dataSource)
throws SQLException {
    Assert.notNull(dataSource, "No DataSource specified");
    ConnectionHolder conHolder = (ConnectionHolder)
TransactionSynchronization
    Manager.getResource(dataSource);
    if (conHolder != null && (conHolder.hasConnection() ||
conHolder.isSynchronized
    WithTransaction())) {
      conHolder.requested();
      if (!conHolder.hasConnection()) {
        logger.debug("Fetching resumed JDBC Connection from
DataSource");
        conHolder.setConnection(dataSource.getConnection());
      }
      return conHolder.getConnection();
    }
    logger.debug("Fetching JDBC Connection from DataSource");
    Connection con = dataSource.getConnection();
    //当前线程支持同步
    if (TransactionSynchronizationManager.isSynchronizationActive()) {
      logger.debug("Registering transaction synchronization for JDBC
Connection");
      //在事务中使用同一数据库连接
      ConnectionHolder holderToUse = conHolder;
```

```
            if (holderToUse == null) {
                holderToUse = new ConnectionHolder(con);
            }
            else {
               holderToUse.setConnection(con);
            }
            //记录数据库连接
            holderToUse.requested();
            TransactionSynchronizationManager.registerSynchronization(
             new ConnectionSynchronization(holderToUse, dataSource));
            holderToUse.setSynchronizedWithTransaction(true);
            if (holderToUse != conHolder) {
                TransactionSynchronizationManager.bindResource(dataSource,
holderToUse);
            }
        }
        return con;
    }
```

在数据库连接方面，Spring主要考虑的是关于事务方面的处理。基于事务处理的特殊性， Spring需要保证线程中的数据库操作都是使用同一个事务连接。

2．应用用户设定的输入参数

```
    protected void applyStatementSettings(Statement stmt) throws
SQLException {
        int fetchSize = getFetchSize();
        if (fetchSize > 0) {
           stmt.setFetchSize(fetchSize);
```

```
        }
        int maxRows = getMaxRows();
        if (maxRows > 0) {
            stmt.setMaxRows(maxRows);
        }
        DataSourceUtils.applyTimeout(stmt, getDataSource(), getQueryTimeout());
    }
```

setFetchSize最主要是为了减少网络交互次数设计的。访问ResultSet时，如果它每次只从服务器上读取一行数据，则会产生大量的开销。setFetchSize的意思是当调用rs.next时，ResultSet会一次性从服务器上取得多少行数据回来，这样在下次rs.next时，它可以直接从内存中获取数据而不需要网络交互，提高了效率。 这个设置可能会被某些 JDBC 驱动忽略，而且设置过大也会造成内存的上升。

setMaxRows将此Statement对象生成的所有ResultSet对象可以包含的最大行数限制设置为给定数。

3．调用回调函数

处理一些通用方法外的个性化处理,也就是PreparedStatementCallback 类型的参数的doInPreparedStatement方法的回调。

4．警告处理

```
protected void handleWarnings(Statement stmt) throws SQLException {
    //当设置为忽略警告时只尝试打印日志
    if (isIgnoreWarnings()) {
        if (logger.isDebugEnabled()) {
            //如果日志开启的情况下打印日志
            SQLWarning warningToLog = stmt.getWarnings();
```

```
                while (warningToLog != null) {
                    logger.debug("SQLWarning ignored: SQL state '" +
warningToLog.
                    getSQLState() + "', error code '" +
                    warningToLog.getErrorCode() + "', message [" +
warningToLog.
                    getMessage() + "]");
                    warningToLog = warningToLog.getNextWarning();
                }
            }
        }
        else {
            handleWarnings(stmt.getWarnings());
        }
    }
```

这里用到了一个类SQLWarning，SQLWarning提供关于数据库访问警告信息的异常。这些警告直接链接到导致报告警告的方法所在的对象。警告可以从Connection、Statement和ResultSet对象中获得。试图在已经关闭的连接上获取警告将导致抛出异常。类似地，试图在已经关闭的语句上或已经关闭的结果集上获取警告也将导致抛出异常。注意，关闭语句时还会关闭它可能生成的结果集。

很多人不是很理解什么情况下会产生警告而不是异常，在这里给读者提示个最常见的警告DataTruncation：DataTruncation直接继承SQLWarning，由于某种原因意外地截断数据值时会以DataTruncation 警告形式报告异常。

对于警告的处理方式并不是直接抛出异常，出现警告很可能会出现数据错误，但是，并不一定会影响程序执行，所以用户可以自己设置处

理警告的方式，如默认的是忽略警告，当出现警告时只打印警告日志，而另一种方式只直接抛出异常。

5．资源释放

数据库的连接释放并不是直接调用了Connection的API中的close方法。考虑到存在事务的情况，如果当前线程存在事务，那么说明在当前线程中存在共用数据库连接，这种情况下直接使用ConnectionHolder中的released方法进行连接数减一，而不是真正的释放连接。

```
public static void releaseConnection(Connection con, DataSource dataSource) {
    try {
        doReleaseConnection(con, dataSource);
    }
    catch (SQLException ex) {
        logger.debug("Could not close JDBC Connection", ex);
    }
    catch (Throwable ex) {
        logger.debug("Unexpected exception on closing JDBC Connection", ex);
    }
}
    public static void doReleaseConnection(Connection con, DataSource dataSource) throws
    SQLException {
    if (con == null) {
        return;
    }
    if (dataSource != null) {
```

//当前线程存在事务的情况下说明存在共用数据库连接直接使用 ConnectionHolder 中的

released方法进行连接数减一而不是真正的释放连接。

ConnectionHolder conHolder = (ConnectionHolder) TransactionSynchronization

Manager.getResource(dataSource);

if (conHolder != null && connectionEquals(conHolder, con)) {

// It's the transactional Connection: Don't close it.

conHolder.released();

return;

}

}

if (!(dataSource instanceof SmartDataSource) || ((SmartDataSource) dataSource). shouldClose(con)) {

logger.debug("Returning JDBC Connection to DataSource");

con.close();

}

}

8.2.2 Update中的回调函数

PreparedStatementCallback作为一个接口，其中只有一个函数doInPreparedStatement，这个函数是用于调用通用方法execute的时候无法处理的一些个性化处理方法，在update中的函数实现：

public Integer doInPreparedStatement(PreparedStatement ps) throws SQLException {

try {

if (pss != null) {

```
            //设置PreparedStatement所需的全部参数。
            pss.setValues(ps);
          }
          int rows = ps.executeUpdate();
          if (logger.isDebugEnabled()) {
             logger.debug("SQL update affected " + rows + " rows");
          }
          return rows;
      }
      finally {
          if (pss instanceof ParameterDisposer) {
             ((ParameterDisposer) pss).cleanupParameters();
          }
      }
  }
```

其中用于真正执行SQL的ps.executeUpdate没有太多需要讲解的，因为我们平时在直接使用JDBC方式进行调用的时候会经常使用此方法。但是，对于设置输入参数的函数pss.setValues (ps)，我们有必要去深入研究一下。在没有分析源码之前，我们至少可以知道其功能，不妨再回顾下Spring中使用SQL的执行过程，直接使用：

```
jdbcTemplate.update("insert into user(name,age,sex)values(?,?,?)",
new Object[] { user.getName(), user.getAge(),
user.getSex() }, new int[] { java.sql.Types.VARCHAR,
java.sql.Types.INTEGER, java.sql.Types.VARCHAR });
```

SQL语句对应的参数，对应参数的类型清晰明了，这都归功于Spring为我们做了封装，而真正的JDBC调用其实非常繁琐，你需要这么做：

PreparedStatement updateSales = con.prepareStatement("insert into user(name,age,
sex)values(?,?,?)");
updateSales.setString(1, user.getName());
updateSales.setInt(2, user.getAge());
updateSales.setString(3, user.getSex());

那么看看Spring是如何做到封装上面的操作呢？

首先，所有的操作都是以pss.setValues(ps)为入口的。还记得我们之前的分析路程吗？这个pss所代表的当前类正是ArgPreparedStatementSetter。其中的setValues如下：

```
public void setValues(PreparedStatement ps) throws SQLException {
    int parameterPosition = 1;
    if (this.args != null) {
      //遍历每个参数以作类型匹配及转换
      for (int i = 0; i < this.args.length; i++) {
        Object arg = this.args[i];
        //如果是集合类则需要进入集合类内部递归解析集合内部属性
        if (arg instanceof Collection && this.argTypes[i] != Types.ARRAY) {
          Collection entries = (Collection) arg;
          for (Iterator it = entries.iterator(); it.hasNext();) {
            Object entry = it.next();
            if (entry instanceof Object[]) {
              Object[] valueArray = ((Object[])entry);
              for (int k = 0; k < valueArray.length; k++) {
                Object argValue = valueArray[k];
```

```
                    doSetValue(ps, parameterPosition, this.argTypes[i],
                    argValue);
                    parameterPosition++;
                  }
                }else {
                  doSetValue(ps, parameterPosition, this.argTypes[i],
entry);
                  parameterPosition++;
                }
              }
            }else {
              //解析当前属性
              doSetValue(ps, parameterPosition, this.argTypes[i], arg);
              parameterPosition++;
            }
          }
        }
    }
```

对单个参数及类型的匹配处理：

```
    protected void doSetValue(PreparedStatement ps, int parameterPosition,
int argType, Object argValue)
    throws SQLException {
        StatementCreatorUtils.setParameterValue(ps, parameterPosition,
argType, argValue);
    }
    public static void setParameterValue(
    PreparedStatement ps, int paramIndex, int sqlType, Object inValue)
```

```
    throws SQLException {
        setParameterValueInternal(ps, paramIndex, sqlType, null, null,
inValue);
    }
    private static void setParameterValueInternal(
    PreparedStatement ps, int paramIndex, int sqlType, String typeName,
Integer
    scale, Object inValue)
    throws SQLException {
        String typeNameToUse = typeName;
        int sqlTypeToUse = sqlType;
        Object inValueToUse = inValue;
        if (inValue instanceof SqlParameterValue) {
            SqlParameterValue parameterValue = (SqlParameterValue)
inValue;
            if (logger.isDebugEnabled()) {
                logger.debug("Overriding type info with runtime info from
SqlParameterValue:
                column index " + paramIndex +
                ", SQL type " + parameterValue.getSqlType() +
                ", Type name " + parameterValue.getTypeName());
            }
            if (parameterValue.getSqlType() !=
SqlTypeValue.TYPE_UNKNOWN) {
                sqlTypeToUse = parameterValue.getSqlType();
            }
            if (parameterValue.getTypeName() != null) {
```

```
                    typeNameToUse = parameterValue.getTypeName();
                }
                inValueToUse = parameterValue.getValue();
            }
            if (logger.isTraceEnabled()) {
                logger.trace("Setting SQL statement parameter value: column
index " +
                paramIndex +
                ", parameter value [" + inValueToUse +
                "], value class [" + (inValueToUse != null ? inValueToUse.
                getClass().getName() : "null") +
                "], SQL type " + (sqlTypeToUse ==
SqlTypeValue.TYPE_UNKNOWN ?
                "unknown" : Integer.toString(sqlTypeToUse)));
            }
            if (inValueToUse == null) {
                setNull(ps, paramIndex, sqlTypeToUse, typeNameToUse);
            }
            else {
                setValue(ps, paramIndex, sqlTypeToUse, typeNameToUse, scale,
inValueToUse);
            }
        }
```

8.3 query功能的实现

在之前的章节中我们介绍了update方法的功能实现，那么在数据库

操作中查找操作也是使用率非常高的函数，同样我们也需要了解它的实现过程。使用方法如下：

```
List<User> list = jdbcTemplate.query("select * from user where
age=?",new
Object[]{20},new int[]{java.sql.Types.INTEGER} ,new
UserRowMapper());
```

跟踪jdbcTemplate中的query方法。

```
public <T> List<T> query(String sql, Object[] args, int[] argTypes,
RowMapper<T> rowMapper)
throws DataAccessException {
    return query(sql, args, argTypes, new
RowMapperResultSetExtractor<T>
   (rowMapper));
}
public <T> T query(String sql, Object[] args, int[] argTypes,
ResultSetExtractor<T> rse)
throws DataAccessException {
    return query(sql, newArgTypePreparedStatementSetter(args,
argTypes), rse);
}
```

上面函数中与update方法中都同样使用了newArgTypePreparedStatementSetter。

```
public <T> T query(String sql, PreparedStatementSetter pss,
ResultSetExtractor<T> rse)
throws DataAccessException {
    return query(new SimplePreparedStatementCreator(sql), pss, rse);
}
```

```
    public <T> T query(
    PreparedStatementCreator psc, final PreparedStatementSetter pss, final
    ResultSetExtractor<T> rse)
    throws DataAccessException {
        Assert.notNull(rse, "ResultSetExtractor must not be null");
        logger.debug("Executing prepared SQL query");
        return execute(psc, new PreparedStatementCallback<T>() {
            public T doInPreparedStatement(PreparedStatement ps) throws
SQLException {
                ResultSet rs = null;
                try {
                    if (pss != null) {
                        pss.setValues(ps);
                    }
                    rs = ps.executeQuery();
                    ResultSet rsToUse = rs;
                    if (nativeJdbcExtractor != null) {
                        rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
                    }
                    return rse.extractData(rsToUse);
                }
                finally {
                    JdbcUtils.closeResultSet(rs);
                    if (pss instanceof ParameterDisposer) {
                        ((ParameterDisposer) pss).cleanupParameters();
                    }
                }
```

```
        }
    });
}
```

可以看到整体套路与update差不多的，只不过在回调类PreparedStatementCallback的实现中使用的是ps.executeQuery()执行查询操作，而且在返回方法上也做了一些额外的处理。

rse.extractData(rsToUse)方法负责将结果进行封装并转换至 POJO，rse 当前代表的类为RowMapperResultSetExtractor，而在构造RowMapperResultSetExtractor 的时候我们又将自定义的rowMapper设置了进去。调用代码如下：

```
public List<T> extractData(ResultSet rs) throws SQLException {
     List<T> results = (this.rowsExpected > 0 ? new ArrayList<T>
(this.rowsExpected) :
    new ArrayList<T>());
     int rowNum = 0;
     while (rs.next()) {
         results.add(this.rowMapper.mapRow(rs, rowNum++));
    }
     return results;
}
```

上面的代码中并没有什么复杂的逻辑，只是对返回结果遍历并以此使用rowMapper进行转换。

之前讲了update方法以及query方法，使用这两个函数示例的SQL都是带有参数的，也就是带有“？”的，那么还有另一种情况是不带有“?”的，Spring中使用的是另一种处理方式。例如：

```
List<User> list = jdbcTemplate.query("select * from user", new
UserRowMapper());
```

跟踪进入：

```
public <T> List<T> query(String sql, RowMapper<T> rowMapper)
throws DataAccessException {
    return query(sql, new RowMapperResultSetExtractor<T>
(rowMapper));
}
public <T> T query(final String sql, final ResultSetExtractor<T> rse)
throws
DataAccessException {
    Assert.notNull(sql, "SQL must not be null");
    Assert.notNull(rse, "ResultSetExtractor must not be null");
    if (logger.isDebugEnabled()) {
        logger.debug("Executing SQL query [" + sql + "]");
    }
    class QueryStatementCallback implements StatementCallback<T>,
SqlProvider {
        public T doInStatement(Statement stmt) throws SQLException {
            ResultSet rs = null;
            try {
                rs = stmt.executeQuery(sql);
                ResultSet rsToUse = rs;
                if (nativeJdbcExtractor != null) {
                    rsToUse = nativeJdbcExtractor.getNativeResultSet(rs);
                }
                return rse.extractData(rsToUse);
            }
            finally {
```

```
                JdbcUtils.closeResultSet(rs);
            }
        }
        public String getSql() {
            return sql;
        }
    }
    return execute(new QueryStatementCallback());
}
```

与之前的 query 方法最大的不同是少了参数及参数类型的传递，自然也少了 Prepared StatementSetter类型的封装。既然少了PreparedStatementSetter类型的传入，调用的execute方法自然也会有所改变了。

```
public <T> T execute(StatementCallback<T> action) throws
DataAccessException {
        Assert.notNull(action, "Callback object must not be null");
        Connection con = DataSourceUtils.getConnection(getDataSource());
        Statement stmt = null;
        try {
            Connection conToUse = con;
            if (this.nativeJdbcExtractor != null &&
            this.nativeJdbcExtractor.isNativeConnectionNecessary ForNative
            Statements()) {
                conToUse =
this.nativeJdbcExtractor.getNativeConnection(con);
            }
            stmt = conToUse.createStatement();
```

```
            applyStatementSettings(stmt);
             Statement stmtToUse = stmt;
             if (this.nativeJdbcExtractor != null) {
                 stmtToUse = this.nativeJdbcExtractor.getNativeStatement(stmt);
            }
             T result = action.doInStatement(stmtToUse);
            handleWarnings(stmt);
             return result;
        }
         catch (SQLException ex) {
             // Release Connection early, to avoid potential connection pool
deadlock
             // in the case when the exception translator hasn't been initialized
yet.
            JdbcUtils.closeStatement(stmt);
             stmt = null;
             DataSourceUtils.releaseConnection(con, getDataSource());
             con = null;
             throw getExceptionTranslator().translate("StatementCallback",
getSql(action), ex);
        }
         finally {
            JdbcUtils.closeStatement(stmt);
             DataSourceUtils.releaseConnection(con, getDataSource());
        }
    }
```

这个exexute与之前的execute并无太大差别，都是做一些常规的处

理，诸如获取连接、释连接等，但是，有一个地方是不一样的，就是statement的创建。这里直接使用connection创建，而带有参数的SQL使用的是PreparedStatementCreator类来创建的。一个是普通的Statement，另一个是PreparedStatement，两者究竟是何区别呢？

PreparedStatement接口继承Statement，并与之在两方面有所不同。

PreparedStatement实例包含已编译的SQL语句。这就是使语句“准备好”。包含于PreparedStatement对象中的SQL 语句可具有一个或多个IN参数。IN参数的值在SQL 语句创建时未被指定。相反的，该语句为每个IN参数保留一个问号（"?"）作为占位符。每个问号的值必须在该语句执行之前，通过适当的setXXX方法来提供。

由于PreparedStatement对象已预编译过，所以其执行速度要快于Statement对象。因此，多次执行的SQL语句经常创建为PreparedStatement对象，以提高效率。

作为Statement的子类，PreparedStatement继承了Statement的所有功能。另外，它还添加了一整套方法，用于设置发送给数据库以取代IN参数占位符的值。同时，三种方法execute、executeQuery和executeUpdate已被更改以使之不再需要参数。这些方法的Statement形式（接受SQL语句参数的形式）不应该用于PreparedStatement对象。

8.4 queryForObject

Spring中不仅仅为我们提供了query方法，还在此基础上做了封装，提供了不同类型的query方法，如图8-1所示。

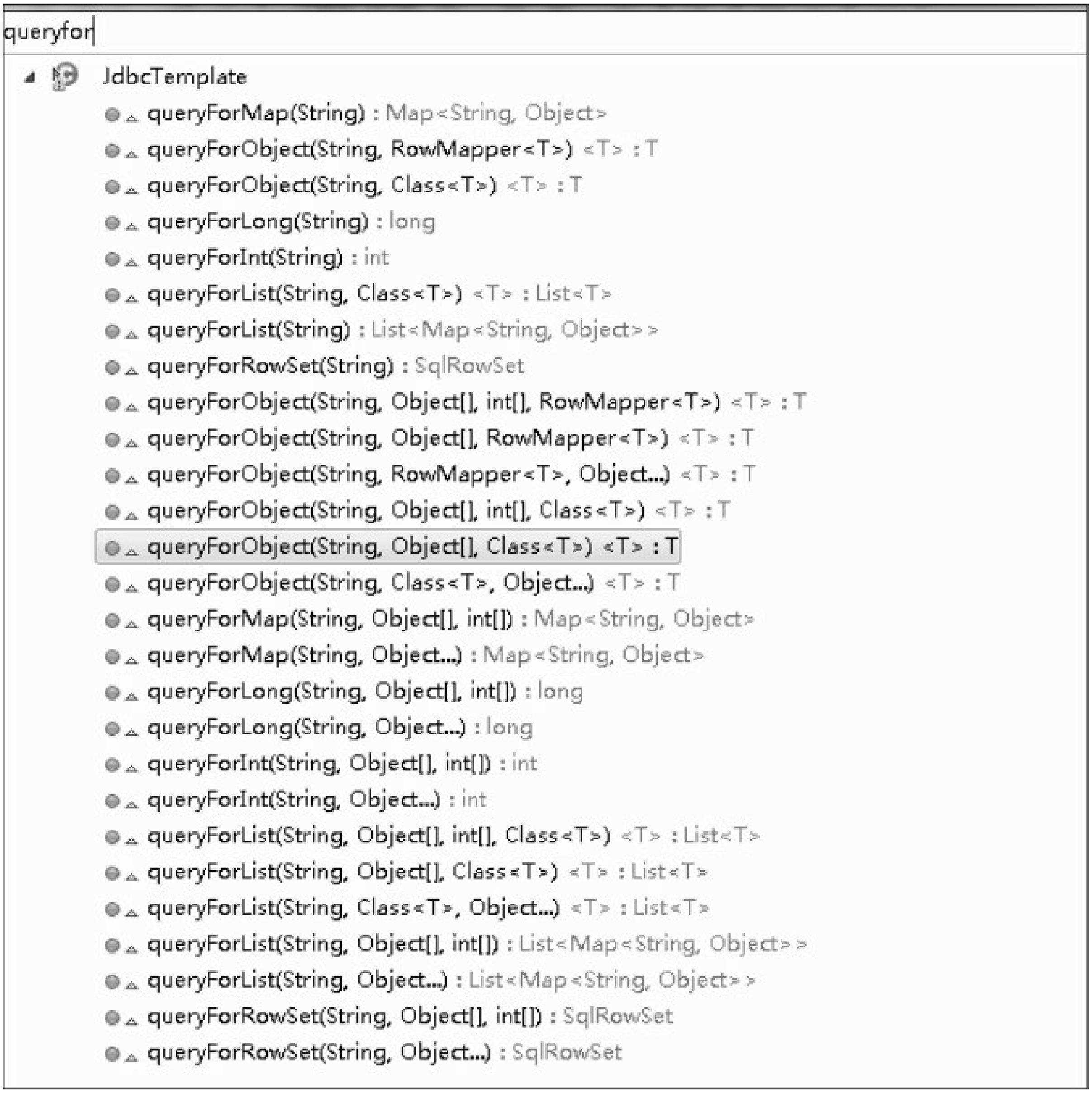


图8-1 Spring中的query相关方法

我们以queryForObject为例，来讨论一下Spring是如何在返回结果的基础上进行封装的。

```
public <T> T queryForObject(String sql, Class<T> requiredType)
throws DataAccessException {
    return queryForObject(sql,
getSingleColumnRowMapper(requiredType));
```

```
}
public <T> T queryForObject(String sql, RowMapper<T> rowMapper)
throws DataAccessException {
    List<T> results = query(sql, rowMapper);
    return DataAccessUtils.requiredSingleResult(results);
}
```

其实最大的不同还是对于RowMapper的使用。SingleColumnRowMapper类中的mapRow：

```
public T mapRow(ResultSet rs, int rowNum) throws SQLException {
   //验证返回结果数
    ResultSetMetaData rsmd = rs.getMetaData();
    int nrOfColumns = rsmd.getColumnCount();
    if (nrOfColumns != 1) {
        throw new IncorrectResultSetColumnCountException(1,
nrOfColumns);
    }
   //抽取第一个结果进行处理
    Object result = getColumnValue(rs, 1, this.requiredType);
    if (result != null && this.requiredType != null &&
!this.requiredType.
    isInstance(result)) {
      //转换到对应的类型
        try {
            return (T) convertValueToRequiredType(result,
this.requiredType);
        }
        catch (IllegalArgumentException ex) {
```

throw new TypeMismatchDataAccessException(
"Type mismatch affecting row number " + rowNum + " and
column
type '" +
rsmd.getColumnTypeName(1) + "': " + ex.getMessage());
}
}
return (T) result;
}

对应的类型转换函数：

protected Object convertValueToRequiredType(Object value, Class
requiredType) {
if (String.class.equals(requiredType)) {
return value.toString();
}
else if (Number.class.isAssignableFrom(requiredType)) {
if (value instanceof Number) {
// Convert original Number to target Number class.
//转换原始Number类型的实体到Number类
return NumberUtils.convertNumberToTargetClass(((Number)
value),
requiredType);
}else {
//转换string类型的值到Number类
return NumberUtils.parseNumber(value.toString(),
requiredType);
}

```
        }
        else {
            throw new IllegalArgumentException(
            "Value [" + value + "] is of type [" + value.getClass().getName() +
            "] and cannot be converted to required type [" + requiredType.
getName() + "]");
        }
    }
```

第9章　整合MyBatis

MyBatis本是Apache的一个开源项目iBatis，2010年这个项目由Apache Software Foundation迁移到了Google Code，并且改名为MyBatis（下载地址为http://code.google.com/p/mybatis/）。

MyBatis是支持普通 SQL查询、存储过程和高级映射的优秀持久层框架。MyBatis消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索。MyBatis使用简单的XML或注解用于配置和原始映射，将接口和Java的POJOs（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录。

9.1 MyBatis独立使用

尽管我们接触更多的是MyBatis与Spring的整合使用，但是MyBatis有它自己的独立使用方法，了解其独立使用的方法套路对分析Spring整合MyBatis非常有帮助，因为Spring无非就是将这些功能进行封装以简化我们的开发流程。MyBatis独立使用包括以下几步。

（1）建立PO。

用于对数据库中数据的映射，使程序员更关注于对Java类的使用而不是数据库的操作。

```
public class User {
private Integer id;
    private String name;
    private Integer age;
      //省略set/get方法
    public User(String name, Integer age) {
super();
    this.name = name;
    this.age = age;
    }
    public User() {
      super();
    } //必须要有这个无参构造方法，不然根据UserMapper.xml中的配置，在查询数据库时，将不能反射构造出
   User实例
}
```

（2）建立Mapper。

数据库操作的映射文件，也就是我们常常说的 DAO，用于映射数据库的操作，可以通过配置文件指定方法对应的SQL语句或者直接使用Java提供的注解方式进行SQL指定。

```
public interface UserMapper {
    public void insertUser(User user);
    public User getUser(Integer id);
}
```

（3）建立配置文件。

配置文件主要用于配置程序中可变性高的设置，一个偏大的程序一定会存在一些经常会变化的变量，如果每次变化都需要改变源码那会是非常糟糕的设计，所以，我们看到各种各样的框架或者应用的时候都免不了要配置配置文件，MyBatis 中的配置文件主要封装在 configuration 中，配置文件的基本结构如图9-1所示。

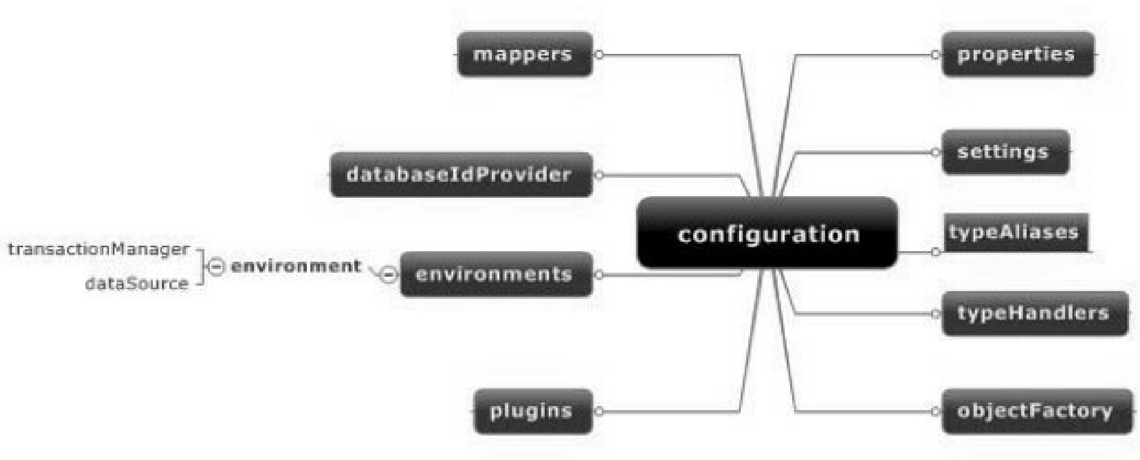


图9-1 配置文件结构

configuration：根元素。

properties：定义配置外在化。

settings：一些全局性的配置。

typeAliases：为一些类定义别名。

typeHandlers：定义类型处理，也就是定义 Java 类型与数据库中的数据类型之间的转换关系。

objectFactory：用于指定结果集对象的实例是如何创建的。

plugins：MyBatis的插件，插件可以修改MyBatis内部的运行规则。

environments：环境。

environment：配置MyBatis的环境。

transactionManager：事务管理器。

dataSource：数据源。

mappers：指定映射文件或映射类。

读者如果对上面的各个配置具体使用方法感兴趣，可以进一步查阅相关资料，这里只举出最简单的实例以方便读者快速回顾MyBatis。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <settings>
    <!-- changes from the defaults for testing -->
    <setting name="cacheEnabled" value="false" />
    <setting name="useGeneratedKeys" value="true" />
    <setting name="defaultExecutorType" value="REUSE" />
  </settings>
  <typeAliases>
    <typeAlias alias="User" type="bean.User"/>
  </typeAliases>
  <environments default="development">
    <environment id="development">
      <transactionManager type="jdbc"/>
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://localhost/lexueba"/>
        <property name="username" value="root"/>
        <property name="password" value="haojia0421xixi"/>
```

```
        </dataSource>
      </environment>
    </environments>
    <mappers>
       <mapper resource="resource/UserMapper.xml" />
    </mappers>
</configuration>
```

（4）建立映射文件。

对应于 MyBaits 全局配置中的 mappers 配置属性，主要用于建立对应数据库操作接口的SQL映射。MyBatis会将这里设定的SQL与对应的Java接口相关联，以保证在MyBatis中调用接口的时候会到数据库中执行相应的SQL来简化开发。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
   PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="Mapper.UserMapper">
<!-- 这里 namespace 必须是 UserMapper 接口的路径，不然要运行的时候要报错 “is not known to the
MapperRegistry”-->
   <insert id="insertUser" parameterType="User" >
      insert into user(name,age) values(#{name},#{age})
      <!--这里sql结尾不能加分号，否则报“ORA-00911”的错误 -->
   </insert>
   <!-- 这里的id必须和UserMapper接口中的接口方法名相同，不然运行的时候也要报错 -->
   <select id="getUser" resultType="User"
```

```
parameterType="java.lang.Integer" >
        select * from user where id=#{id}
    </select>
</mapper>
```

（5）建立测试类。

至此我们已经完成了MyBatis的建立过程，接下来的工作就是对之前的所有工作进行测试，以便直接查看MyBatis为我们提供的效果。

```
public class MyBatisUtil {
    private final static SqlSessionFactory sqlSessionFactory;
    static {
        String resource = "resource/mybatis-config.xml";
        Reader reader = null;
        try {
            reader = Resources.getResourceAsReader(resource);
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        sqlSessionFactory = new
SqlSessionFactoryBuilder().build(reader);
    }
    public static SqlSessionFactory getSqlSessionFactory() {
        return sqlSessionFactory;
    }
}
public class TestMapper {
    static SqlSessionFactory sqlSessionFactory = null;
    static {
```

```
        sqlSessionFactory = MyBatisUtil.getSqlSessionFactory();
    }
    @Test
     public void testAdd() {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        try {
            UserMapper userMapper =
sqlSession.getMapper(UserMapper.class);
            User user = new User("tom",new Integer(5));
           userMapper.insertUser(user);
           sqlSession.commit();//这里一定要提交，不然数据进不去数据
库中
        } finally {
           sqlSession.close();
        }
    }
    @Test
     public void getUser() {
        SqlSession sqlSession = sqlSessionFactory.openSession();
        try {
            UserMapper userMapper =
sqlSession.getMapper(UserMapper.class);
            User user = userMapper.getUser(1);
            System.out.println("name: "+user.getName()+"|age:
"+user.getAge());
        } finally {
           sqlSession.close();
```

```
        }
    }
}
```

注意，这里在数据库设定了id自增策略，所以插入的数据会直接在数据库中赋值，当执行测试后如果数据表为空，那么在表中会出现一条我们插入的数据，并会在查询时将此数据查出。

9.2 Spring整合MyBatis

了解了MyBatis的独立使用过程后，我们再看看它与Spring整合的使用方式，比对之前的示例来找出 Spring 究竟为我们做了哪些操作来简化程序员的业务开发。由于在上面示例基础上作更改，所以，User与UserMapper保持不变。

（1）Spring配置文件。

配置文件是Spring的核心，Spring的所有操作也都是由配置文件开始的，所以，我们的示例也首先从配置文件开始。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www.Springframework.
     org/ schema/beans/Spring-beans-3.0.xsd">
     <bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName"
value="com.mysql.jdbc.Driver"></property>
```

```
            <property name="url"
value="jdbc:mysql://localhost:3306/lexueba?useUnicode=
           true&amp;characterEncoding=UTF-
8&amp;zeroDateTimeBehavior=convertToNull"></property>
            <property name="username" value="root"></property>
            <property name="password" value="haojia0421xixi"></property>
            <property name="maxActive" value="100"></property>
            <property name="maxIdle" value="30"></property>
            <property name="maxWait" value="500"></property>
            <property name="defaultAutoCommit" value="true"></property>
        </bean>
         <bean id="sqlSessionFactory"
class="org.mybatis.Spring.SqlSessionFactoryBean">
            <property name="configLocation"
value="classpath:test/mybatis/MyBatis-Configuration.
           xml"></property>
            <property name="dataSource" ref="dataSource" />
        </bean>
         <bean id="userMapper"
class="org.mybatis.Spring.mapper.MapperFactoryBean">
            <property name="mapperInterface"
value="test.mybatis.dao.UserMapper"></property>
            <property name="sqlSessionFactory" ref="sqlSessionFactory">
</property>
        </bean>
    </beans>
```

对比之前独立使用 MyBatis 的配置文件，我们发现，之前在

environments 中设置的dataSource 被转移到了 Spring 的核心配置文件中管理。而且，针对于 MyBatis ，注册了 org.mybatis.Spring.SqlSessionFactoryBean 类型 bean，以及用于映射接口的 org.mybatis.Spring. mapper.MapperFactoryBean，这两个bean的作用我们会在稍后分析。

之前我们了解到，MyBatis 提供的配置文件包含了诸多属性，虽然大多数情况我们都会保持MyBatis原有的风格，将MyBatis的配置文件独立出来，并在Spring中的org.mybatis.Spring. SqlSessionFactoryBean类型的bean中通过configLocation属性引入，但是，这并不代表Spring不支持直接配置。以上面示例为例，你完全可以省去MyBatis-Configuration.xml，而将其中的配置以属性的方式注入到SqlSessionFactoryBean中，至于每个属性名称以及用法，我们会在后面的章节中进行详细的分析。

（2）MyBatis配置文件。

对比独立使用MyBatis时的配置文件，当前的配置文件除了移除environments配置外并没有太多的变化。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <typeAliases>
       <typeAlias alias="User" type="test.mybatis.bean.User"/>
   </typeAliases>
   <mappers>
       <mapper resource="test/mybatis/UserMapper.xml"/>
   </mappers>
```

```
</configuration>
```

（3）映射文件（保持不变）。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/
dtd/mybatis-3-mapper.dtd">
<mapper namespace="test.mybatis.dao.UserMapper">
    <insert id="insertUser" parameterType="User" >
        insert into user(name,age) values(#{name},#{age})
    </insert>
    <select id="getUser" resultType="User" parameterType="java.lang.String" >
        select * from user where name=#{name}
    </select>
</mapper>
```

（4）测试。

至此，我们已经完成了Spring与MyBatis的整合，我们发现，对于MyBatis方面的配置文件，除了将dataSource配置移到Spring配置文件中管理外，并没有太多变化，而在Spring的配置文件中又增加了用于处理MyBatis的两个bean。

Spring整合MyBatis的优势主要在于使用上，我们来看看Spring中使用MyBatis的用法。

```
public class UserServiceTest {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("test/ mybatis/
        applicationContext.xml");
```

```
            UserMapper userDao =
(UserMapper)context.getBean("userMapper");
            System.out.println(userDao.getUser("1"));
        }
    }
```

测试中我们看到，在 Spring 中使用 MyBatis 非常方便，用户甚至无法察觉自己正在使用MyBatis，而这一切相对于独立使用 MyBatis 时必须要做的各种冗余操作来说无非是大大简化了我们的工作量。

9.3 源码分析

通过Spring整合MyBatis的示例，我们感受到了Spring为用户更加快捷地进行开发所做的努力，开发人员的工作效率由此得到了显著的提升。但是，相对于使用来说，我们更想知道其背后所隐藏的秘密，Spring整合MyBatis是何如实现的呢？通过分析整合示例中的配置文件，我们可以知道配置的 bean 其实是成树状结构的，而在树的最顶层是类型为 org.mybatis. Spring.SqlSessionFactoryBean的bean，它将其他相关bean组装在了一起，那么，我们的分析就从此类开始。

9.3.1 sqlSessionFactory创建

通过配置文件我们分析，对于配置文件的读取解析，Spring 应该通过 org.mybatis.Spring. SqlSessionFactoryBean封装了MyBatis中的实现。我们进入这个类，首先查看这个类的层次结构，如图9-2所示。根据这个类的层次结构找出我们感兴趣的两个接口， FactoryBean和InitializingBean。



图9-2 SqlSessionFactoryBean类的层次结构图

InitializingBean：实现此接口的bean会在初始化时调用其afterPropertiesSet方法来进行bean的逻辑初始化。

FactoryBean：一旦某个bean实现次接口，那么通过getBean方法获取bean时其实是获取此类的getObject()返回的实例。

我们首先以InitializingBean接口的afterPropertiesSet()方法作为突破点。

1．SqlSessionFactoryBean的初始化

查看org.mybatis.Spring.SqlSessionFactoryBean类型的bean在初始化时做了哪些逻辑实现。

```
public void afterPropertiesSet() throws Exception {
    notNull(dataSource, "Property 'dataSource' is required");
    notNull(sqlSessionFactoryBuilder, "Property
'sqlSessionFactoryBuilder' is required");
    this.sqlSessionFactory = buildSqlSessionFactory();
}
```

很显然，此函数主要目的就是对于sqlSessionFactory的初始化，通过之前展示的独立使用MyBatis的示例，我们了解到SqlSessionFactory是所有MyBatis功能的基础。

```
protected SqlSessionFactory buildSqlSessionFactory() throws
```

```
IOException {
            Configuration configuration;
            XMLConfigBuilder xmlConfigBuilder = null;
            if (this.configLocation != null) {
                xmlConfigBuilder = new
XMLConfigBuilder(this.configLocation.getInputStream(), null,
               this.configurationProperties);
                configuration = xmlConfigBuilder.getConfiguration();
            } else {
                if (this.logger.isDebugEnabled()) {
                    this.logger.debug("Property 'configLocation' not specified, using
default MyBatis
                   Configuration");
               }
                configuration = new Configuration();
               configuration.setVariables(this.configurationProperties);
            }
            if (this.objectFactory != null) {
               configuration.setObjectFactory(this.objectFactory);
            }
            if (this.objectWrapperFactory != null) {

configuration.setObjectWrapperFactory(this.objectWrapperFactory);
            }
            if (hasLength(this.typeAliasesPackage)) {
                String[] typeAliasPackageArray =
tokenizeToStringArray(this.typeAliasesPackage,
```

```
ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
        for (String packageToScan : typeAliasPackageArray) {

configuration.getTypeAliasRegistry().registerAliases(packageToScan,
            typeAliasesSuperType == null ? Object.class :
typeAliasesSuperType);
            if (this.logger.isDebugEnabled()) {
              this.logger.debug("Scanned package: '" + packageToScan + "'
for aliases");
            }
        }
      }
      if (!isEmpty(this.typeAliases)) {
        for (Class<?> typeAlias : this.typeAliases) {
          configuration.getTypeAliasRegistry().registerAlias(typeAlias);
            if (this.logger.isDebugEnabled()) {
              this.logger.debug("Registered type alias: '" + typeAlias + "'");
            }
        }
      }
      if (!isEmpty(this.plugins)) {
        for (Interceptor plugin : this.plugins) {
        configuration.addInterceptor(plugin);
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Registered plugin: '" + plugin + "'");
        }
```

```
      }
    }
    if (hasLength(this.typeHandlersPackage)) {
        String[] typeHandlersPackageArray =
tokenizeToStringArray(this.typeHandlersPackage,

ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS);
        for (String packageToScan : typeHandlersPackageArray) {
          configuration.getTypeHandlerRegistry().register(packageToScan);
           if (this.logger.isDebugEnabled()) {
              this.logger.debug("Scanned package: '" + packageToScan + "'
for type handlers");
          }
      }
    }
    if (!isEmpty(this.typeHandlers)) {
        for (TypeHandler<?> typeHandler : this.typeHandlers) {
          configuration.getTypeHandlerRegistry().register(typeHandler);
           if (this.logger.isDebugEnabled()) {
              this.logger.debug("Registered type handler: '" + typeHandler +
"'");
          }
      }
    }
    if (xmlConfigBuilder != null) {
        try {
          xmlConfigBuilder.parse();
```

```
                if (this.logger.isDebugEnabled()) {
                    this.logger.debug("Parsed configuration file: '" +
this.configLocation + "'");
                }
            } catch (Exception ex) {
                throw new NestedIOException("Failed to parse config resource: "
+ this.
                configLocation, ex);
            } finally {
               ErrorContext.instance().reset();
            }
        }
        if (this.transactionFactory == null) {
            this.transactionFactory = new SpringManagedTransactionFactory();
        }
        Environment environment = new Environment(this.environment,
this.transactionFactory,
        this.dataSource);
        configuration.setEnvironment(environment);
        if (this.databaseIdProvider != null) {
            try {

configuration.setDatabaseId(this.databaseIdProvider.getDatabaseId
(this.dataSource));
            } catch (SQLException e) {
                throw new NestedIOException("Failed getting a databaseId", e);
            }
```

```
        }
        if (!isEmpty(this.mapperLocations)) {
            for (Resource mapperLocation : this.mapperLocations) {
                if (mapperLocation == null) {
                    continue;
                }
                try {
                    XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(mapperLocation.
                    getInputStream(),
                    configuration, mapperLocation.toString(),
configuration.getSqlFragments());
                    xmlMapperBuilder.parse();
                } catch (Exception e) {
                    + "'", e);
                    throw new NestedIOException("Failed to parse mapping
resource: '" + mapperLocation
                } finally {
                    ErrorContext.instance().reset();
                }
                if (this.logger.isDebugEnabled()) {
                    this.logger.debug("Parsed mapper file: '" + mapperLocation +
"'");
                }
            }
        } else {
            if (this.logger.isDebugEnabled()) {
```

```
            this.logger.debug("Property 'mapperLocations' was not specified or
no matching
            resources found");
        }
    }
    return this.sqlSessionFactoryBuilder.build(configuration);
}
```

从函数中可以看到，尽管我们还是习惯于将MyBatis的配置与Spring的配置独立出来，但是，这并不代表Spring中的配置不支持直接配置。也就是说，在上面提供的示例中，你完全可以取消配置中的configLocation属性，而把其中的属性直接写在SqlSessionFactoryBean中。

```
<bean id="sqlSessionFactory"
class="org.mybatis.Spring.SqlSessionFactoryBean">
    <property name="configLocation"
value="classpath:test/mybatis/MyBatis- Configuration.
    xml"></property>
     <property name="dataSource" ref="dataSource" />
     <property name="typeAliasesPackage" value="aaaaa"/>
     ... ...
</bean>
```

从这个函数中可以得知，配置文件还可以支持其他多种属性的配置，如 configLocation、objectFactory、objectWrapperFactory、typeAliasesPackage、typeAliases、typeHandlersPackage、plugins、typeHandlers、transactionFactory、databaseIdProvider、mapperLocations。

其实，如果只按照常用的配置，那么我们只需要在函数最开始按照

如下方式处理configuration：

```
xmlConfigBuilder = new
XMLConfigBuilder(this.configLocation.getInputStream(), null,
    this.configurationProperties);
      configuration = xmlConfigBuilder.getConfiguration();
```

根据configLocation构造XMLConfigBuilder并进行解析，但是，为了体现Spring更强大的兼容性，Spring还整合了MyBatis中其他属性的注入，并通过实例configuration来承载每一步所获取的信息并最终使用sqlSessionFactoryBuilder 实例根据解析到的 configuration 创建SqlSessionFactory实例。

2．获取**SqlSessionFactoryBean**实例

由于SqlSessionFactoryBean实现了FactoryBean接口，所以当通过getBean方法获取对应实例时，其实是获取该类的 getObject()函数返回的实例，也就是获取初始化后的 sqlSession Factory属性。

```
public SqlSessionFactory getObject() throws Exception {
    if (this.sqlSessionFactory == null) {
      afterPropertiesSet();
    }
    return this.sqlSessionFactory;
}
```

9.3.2 MapperFactoryBean的创建

为了使用 MyBatis 功能，示例中的 Spring 配置文件提供了两个bean，除了之前分析的SqlSsionFactoryBean类型的bean以外，还有一个是MapperFactoryBean类型的bean。

结合两个测试用例综合分析，对于单独使用MyBatis的时候调用数据库接口的方式是:

UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

而在这一过程中，其实是MyBatis在获取映射的过程中根据配置信息为UserMapper 类型动态创建了代理类。而对于Spring的创建方式：

UserMapper userMapper =
(UserMapper)context.getBean("userMapper");

Spring中获取的名为userMapper的bean，其实是与单独使用 MyBatis完成了一样的功能，那么我们可以推断，在 bean 的创建过程中一定是使用了 MyBatis 中的原生方法sqlSession.getMapper(UserMapper.class)进行了再一次封装。结合配置文件，我们把分析目标转向org.mybatis.Spring.mapper. MapperFactoryBean，初步推测其中的逻辑应该在此类中实现。同样，还是首先查看的类层次结构图MapperFactoryBean，如图9-3所示。

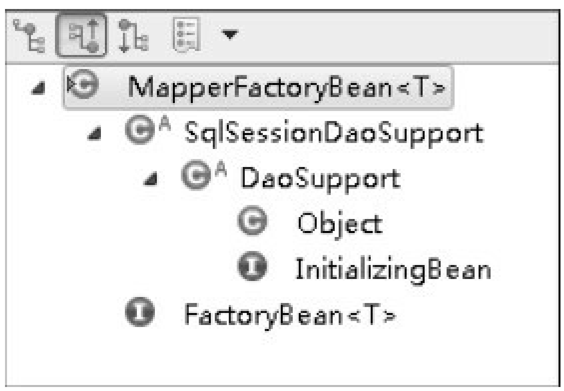


图9-3 MapperFactoryBean类的层次结构图

同样，在实现的接口中发现了我们感兴趣的两个接口InitializingBean与FactoryBean。我们的分析还是从bean的初始化开始。

1．MapperFactoryBean的初始化

因为实现了InitializingBean接口，Spring会保证在bean初始化时首先调用afterPropertiesSet方法来完成其初始化逻辑。追踪父类，发现afterPropertiesSet方法是在DaoSupport类中实现，代码如下：

```
public final void afterPropertiesSet() throws IllegalArgumentException,
BeanInitialization
Exception {
    // Let abstract subclasses check their configuration.
   checkDaoConfig();
    // Let concrete implementations initialize themselves.
    try {
      initDao();
   }
    catch (Exception ex) {
       throw new BeanInitializationException("Initialization of DAO
failed", ex);
   }
}
```

但从函数名称来看我们大体推测，MapperFactoryBean的初始化包括对DAO配置的验证以及对DAO的初始工作，其中initDao()方法是模板方法，设计为留给子类做进一步逻辑处理。而checkDaoConfig()才是我们分析的重点。

```
@Override
    protected void checkDaoConfig() {
      super.checkDaoConfig();
       notNull(this.mapperInterface, "Property 'mapperInterface' is
required");
       Configuration configuration = getSqlSession().getConfiguration();
       if (this.addToConfig &&
!configuration.hasMapper(this.mapperInterface)) {
           try {
```

```
configuration.addMapper(this.mapperInterface);
} catch (Throwable t) {
 logger.error("Error while adding the mapper '" +
this.mapperInterface + "' to
 configuration.", t);
 throw new IllegalArgumentException(t);
} finally {
ErrorContext.instance().reset();
}
}
}
```

super.checkDaoConfig()在SqlSessionDaoSupport类中实现，代码如下：

```
protected void checkDaoConfig() {
 notNull(this.sqlSession, "Property 'sqlSessionFactory' or
'sqlSessionTemplate' are
 required");
}
```

结合代码我们了解到对于DAO配置的验证，Spring做了以下几个方面的工作。

父类中对于sqlSession不为空的验证。

sqlSession作为根据接口创建映射器代理的接触类一定不可以为空，而sqlSession的初始化工作是在设定其sqlSessionFactory属性时完成的。

```
public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory)
{
 if (!this.externalSqlSession) {
```

```
            this.sqlSession = new SqlSessionTemplate(sqlSessionFactory);
        }
    }
```

也就是说，对于下面的配置如果忽略了对于sqlSessionFactory属性的设置，那么在此时就会被检测出来。

```
<bean id="userMapper"
class="org.mybatis.Spring.mapper.MapperFactoryBean">
    <property name="mapperInterface"
value="test.mybatis.dao.UserMapper"></property>
    <property name="sqlSessionFactory" ref="sqlSessionFactory">
</property>
</bean>
```

映射接口的验证。

接口是映射器的基础，sqlSession会根据接口动态创建相应的代理类，所以接口必不可少。

映射文件存在性验证。

对于函数前半部分的验证我们都很容易理解，无非是对配置文件中的属性是否存在做验证，但是后面部分是完成了什么方面的验证呢？如果读者读过 MyBatis 源码，你就会知道，在MyBatis实现过程中并没有手动调用configuration.addMapper方法，而是在映射文件读取过程中一旦解析到如<mapper namespace="Mapper.UserMapper">，便会自动进行类型映射的注册。那么，Spring 中为什么会把这个功能单独拿出来放在验证里呢？这是不是多此一举呢？

在上面的函数中，configuration.addMapper(this.mapperInterface)其实就是将 UserMapper 注册到映射类型中，如果你可以保证这个接口一定存在对应的映射文件，那么其实这个验证并没有必要。但是，由于这个是我们自行决定的配置，无法保证这里配置的接口一定存在对应的映射

文件，所以这里非常有必要进行验证。在执行此代码的时候，MyBatis会检查嵌入的映射接口是否存在对应的映射文件，如果没有回抛出异常，Spring正是在用这种方式来完成接口对应的映射文件存在性验证。

2．获取MapperFactoryBean的实例

由于MapperFactoryBean实现了FactoryBean接口，所以当通过getBean方法获取对应实例的时候其实是获取该类的getObject()函数返回的实例。

```
public T getObject() throws Exception {
    return getSqlSession().getMapper(this.mapperInterface);
}
```

这段代码正是我们在提供 MyBatis 独立使用的时候的一个代码调用。Spring 通过FactoryBean进行了封装。

9.3.3 MapperScannerConfigurer

我们在applicationContext.xml中配置了userMapper供需要时使用。但如果需要用到的映射器较多的话，采用这种配置方式就显得很低效。为了解决这个问题，我们可以使用MapperScanner Configurer，让它扫描特定的包，自动帮我们成批地创建映射器。这样一来，就能大大减少配置的工作量，比如我们将applicationContext.xml文件中的配置改成如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www.Springframework.
    org/schema/beans/Spring-beans-3.0.xsd">
     <bean id="dataSource"
```

```
class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName"
value="com.mysql.jdbc.Driver"></property>
        <property name="url"
value="jdbc:mysql://localhost:3306/lexueba?useUnicode=
        true&amp;characterEncoding=UTF-
8&amp;zeroDateTimeBehavior=convertToNull"></property>
        <property name="username" value="root"></property>
        <property name="password" value="haojia0421xixi"></property>
        <property name="maxActive" value="100"></property>
        <property name="maxIdle" value="30"></property>
        <property name="maxWait" value="500"></property>
        <property name="defaultAutoCommit" value="true"></property>
    </bean>
    <bean id="sqlSessionFactory"
class="org.mybatis.Spring.SqlSessionFactoryBean">
        <property name="configLocation"
value="classpath:test/mybatis/MyBatis- Configuration.
        xml"></property>
        <property name="dataSource" ref="dataSource" />
        <property name="typeAliasesPackage" value="aaaaa"/>
    </bean>
  <!-- 注释掉原有代码
    <bean id="userMapper"
class="org.mybatis.Spring.mapper.MapperFactoryBean">
        <property name="mapperInterface"
value="test.mybatis.dao.UserMapper"></property>
```

```
                <property name="sqlSessionFactory" ref="sqlSessionFactory">
</property>
            </bean>
        -->
            <bean
class="org.mybatis.Spring.mapper.MapperScannerConfigurer">
                <property name="basePackage" value="test.mybatis.dao" />
            </bean>
        </beans>
```

在上面的配置中，我们屏蔽掉了最原始的代码（ userMapper 的创建）而增加了MapperScannerConfigurer 的配置，basePackage 属性是让你为映射器接口文件设置基本的包路径。你可以使用分号或逗号作为分隔符设置多于一个的包路径。每个映射器将会在指定的包路径中递归地被搜索到。被发现的映射器将会使用 Spring 对自动侦测组件默认的命名策略来命名。也就是说，如果没有发现注解，它就会使用映射器的非大写的非完全限定类名。但是如果发现了@Component 或 JSR-330@Named 注解，它会获取名称。

通过上面的配置， Spring 就会帮助我们对 test. mybatis.dao下面的所有接口进行自动的注入，而不需要为每个接口重复在Spring配置文件中进行声明了。那么，这个功能又是如何做到的呢？MapperScanner Configurer中又有哪些核心操作呢？同样，首先查看类的层次结构图，如图9-4所示。

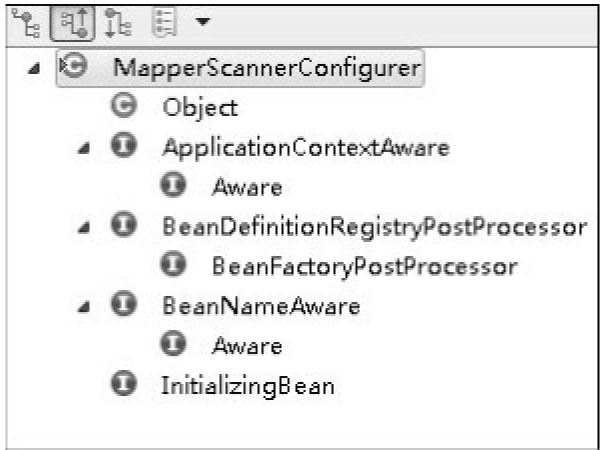


图9-4 MapperScannerConfigurer类的层次结构图

我们又看到了令人感兴趣的接口 InitializingBean，马上查找类的afterPropertiesSet 方法来看看类的初始化逻辑。

```
public void afterPropertiesSet() throws Exception {
    notNull(this.basePackage, "Property 'basePackage' is required");
}
```

很遗憾，分析并没有想我们之前那样顺利，afterPropertiesSet()方法除了一句对basePackage属性的验证代码外并没有太多的逻辑实现。好吧，让我们回过头再次查看 MapperScanner Configurer类层次结构图中感兴趣的接口。于是，我们发现了BeanDefinitionRegistryPostProcessor与BeanFactoryPostProcessor，Spring在初始化的过程中同样会保证这两个接口的调用。

首先查看MapperScannerConfigurer类中对于BeanFactoryPostProcessor接口的实现：

```
public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    // left intentionally blank
}
```

没有任何逻辑实现，只能说明我们找错地方了，继续找，查看MapperScannerConfigurer类中对于BeanDefinitionRegistryPostProcessor接口的实现。

```
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) throws
BeansException {
    if (this.processPropertyPlaceHolders) {
      processPropertyPlaceHolders();
    }
    ClassPathMapperScanner scanner = new
ClassPathMapperScanner(registry);
    scanner.setAddToConfig(this.addToConfig);
    scanner.setAnnotationClass(this.annotationClass);
    scanner.setMarkerInterface(this.markerInterface);
    scanner.setSqlSessionFactory(this.sqlSessionFactory);
    scanner.setSqlSessionTemplate(this.sqlSessionTemplate);

scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);

scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName
    scanner.setResourceLoader(this.applicationContext);
    scanner.setBeanNameGenerator(this.nameGenerator);
    scanner.registerFilters();
    scanner.scan(StringUtils.tokenizeToStringArray(this.basePackage,
Configurable
    ApplicationContext.CONFIG_LOCATION_DELIMITERS));
}
```

Bingo！这次找对地方了。大致看一下代码实现，正是完成了对指定路径扫描的逻辑。那么，我们就以此为入口，详细地分析MapperScannerConfigurer所提供的逻辑实现。

1．processPropertyPlaceHolders属性的处理

首先，难题就是processPropertyPlaceHolders属性的处理。或许读者并未过多接触此属性，我们只能查看processPropertyPlaceHolders()函数来反推此属性所代表的功能。

```
/*
 * BeanDefinitionRegistries are called early in application startup,
before
 * BeanFactoryPostProcessors. This means that
PropertyResourceConfigurers will not have been
 * loaded and any property substitution of this class' properties will
fail. To avoid
 this, find
 * any PropertyResourceConfigurers defined in the context and run
them on this class' bean
 * definition. Then update the values.
 */
private void processPropertyPlaceHolders() {
 Map<String, PropertyResourceConfigurer> prcs =
applicationContext.getBeansOfType
 (PropertyResourceConfigurer.class);
 if (!prcs.isEmpty() && applicationContext instanceof
GenericApplicationContext) {
 BeanDefinition mapperScannerBean =
((GenericApplicationContext) applicationContext)
```

```
            .getBeanFactory().getBeanDefinition(beanName);
            // PropertyResourceConfigurer does not expose any methods to
explicitly perform
            // property placeholder substitution. Instead, create a BeanFactory
that just
            // contains this mapper scanner and post process the factory.
            DefaultListableBeanFactory factory = new
DefaultListableBeanFactory();
            factory.registerBeanDefinition(beanName, mapperScannerBean);
            for (PropertyResourceConfigurer prc : prcs.values()) {
              prc.postProcessBeanFactory(factory);
            }
            PropertyValues values =
mapperScannerBean.getPropertyValues();
            this.basePackage = updatePropertyValue("basePackage", values);
            this.sqlSessionFactoryBeanName =
updatePropertyValue("sqlSessionFactoryBeanName",
            values);
            this.sqlSessionTemplateBeanName =
updatePropertyValue("sqlSessionTemplateBeanName",
            values);
         }
      }
```

不知读者是否悟出了此函数的作用呢？或许此函数的说明会给我们一些提示：BeanDefinitionRegistries会在应用启动的时候调用，并且会早于BeanFactoryPostProcessors的调用，这就意味着PropertyResourceConfigurers还没有被加载所有对于属性文件的引用将会

失效。为避免此种情况发生，此方法手动地找出定义的PropertyResourceConfigurers 并进行提前调用以保证对于属性的引用可以正常工作。

我想读者已经有所感悟，结合之前讲过的PropertyResourceConfigurer的用法，举例说明一下，如要创建配置文件如test.properties，并添加属性对：

```
basePackage=test.mybatis.dao
```

然后在Spring配置文件中加入属性文件解析器：

```
<bean id="mesHandler"
class="org.Springframework.beans.factory.config.Property Placeholder
Configurer">
    <property name="locations">
      <list>
        <value>config/test.properties</value>
      </list>
    </property>
</bean>
```

修改MapperScannerConfigurer类型的bean的定义：

```
<bean class="org.mybatis.Spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="${basePackage}" />
</bean>
```

此时你会发现，这个配置并没有达到预期的效果，因为在解析${basePackage}的时候PropertyPlaceholderConfigurer还没有被调用，也就是属性文件中的属性还没有加载至内存中， Spring还不能直接使用它。为了解决这个问题，Spring提供了processPropertyPlaceHolders属性，你需要这样配置MapperScannerConfigurer类型的bean。

```
<bean class="org.mybatis.Spring.mapper.MapperScannerConfigurer">
```

```
        <property name="basePackage" value="test.mybatis.dao" />
        <property name="processPropertyPlaceHolders" value="true" />
    </bean>
```

通过processPropertyPlaceHolders属性的配置，将程序引入我们正在分析的processProperty PlaceHolders 函数中来完成属性文件的加载。至此，我们终于理清了这个属性的作用，再次回顾这个函数所做的事情。

（1）找到所有已经注册的PropertyResourceConfigurer类型的bean。

（2）模拟 Spring中的环境来用处理器。这里通过使用 new DefaultListableBeanFactory()来模拟 Spring 中的环境（完成处理器的调用后便失效），将映射的 bean，也就是 MapperScanner Configurer类型bean注册到环境中来进行后理器的调用，处理器PropertyPlaceholderConfigurer调用完成的功能，即找出所有 bean中应用属性文件的变量并替换。也就是说，在处理器调用后，模拟环境中模拟的MapperScannerConfigurer类型的bean如果有引入属性文件中的属性那么已经被替换了，这时，再将模拟bean中相关的属性提取出来应用在真实的bean中。

2．根据配置属性生成过滤器

在postProcessBeanDefinitionRegistry方法中可以看到，配置中支持很多属性的设定，但是我们感兴趣的或者说影响扫描结果的并不多，属性设置后通过在 scanner.registerFilters()代码中生成对应的过滤器来控制扫描结果。

```
    public void registerFilters() {
        boolean acceptAllInterfaces = true;
    //对于annotationClass属性的处理
        if (this.annotationClass != null) {
            addIncludeFilter(new AnnotationTypeFilter(this.annotationClass));
            acceptAllInterfaces = false;
```

```
        }
    //对于markerInterface属性的处理
        if (this.markerInterface != null) {
            addIncludeFilter(new AssignableTypeFilter(this.markerInterface)
{
                @Override
                protected boolean matchClassName(String className) {
                    return false;
                }
            });
            acceptAllInterfaces = false;
        }
        if (acceptAllInterfaces) {
            // default include filter that accepts all classes
            addIncludeFilter(new TypeFilter() {
                public boolean match(MetadataReader metadataReader,
MetadataReaderFactory metadata
                ReaderFactory) throws IOException {
                    return true;
                }
            });
        }
    //不扫描package-info.java文件
        addExcludeFilter(new TypeFilter() {
            public boolean match(MetadataReader metadataReader,
MetadataReaderFactory metadata
            ReaderFactory) throws IOException {
```

```
                String className =
metadataReader.getClassMetadata().getClassName();
                return className.endsWith("package-info");
            }
        });
    }
```

代码中得知，根据之前属性的配置生成了对应的过滤器。

（1）annotationClass属性处理。

如果annotationClass不为空，表示用户设置了此属性，那么就要根据此属性生成过滤器以保证达到用户想要的效果，而封装此属性的过滤器就是 AnnotationTypeFilter。Annotation TypeFilter保证在扫描对应Java文件时只接受标记有注解为annotationClass的接口。

（2）markerInterface属性处理。

如果markerInterface不为空，表示用户设置了此属性，那么就要根据此属性生成过滤器以保证达到用户想要的效果，而封装此属性的过滤器就是实现 AssignableTypeFilter 接口的局部类。表示扫描过程中只有实现markerInterface接口的接口才会被接受。

（3）全局默认处理。

在上面两个属性中如果存在其中任何属性，acceptAllInterfaces的值将会变改变，但是如果用户没有设定以上两个属性，那么，Spring会为我们增加一个默认的过滤器实现TypeFilter接口的局部类，旨在接受所有接口文件。

（4）package-info.java处理。

对于命名为 package-info 的 Java文件，默认不作为逻辑实现接口，将其排除掉，使用TypeFilter接口的局部类实现match方法。

从上面的函数我们看出，控制扫描文件Spring通过不同的过滤器完成，这些定义的过滤器记录在了includeFilters和excludeFilters属性中。

```
public void addIncludeFilter(TypeFilter includeFilter) {
   this.includeFilters.add(includeFilter);
}
public void addExcludeFilter(TypeFilter excludeFilter) {
    this.excludeFilters.add(0, excludeFilter);
}
```

至于过滤器为什么会在扫描过程中起作用，我们在讲解扫描实现时候再继续深入研究。

3．扫描**Java**文件

设置了相关属性以及生成了对应的过滤器后便可以进行文件的扫描了，扫描工作是由ClassPathMapperScanner类型的实例scanner中的scan方法完成的。

```
public int scan(String... basePackages) {
    int beanCountAtScanStart = this.registry.getBeanDefinitionCount();
   doScan(basePackages);
//如果配置了includeAnnotationConfig，则注册对应注解的处理器以保证注解功能的正常使用。
    if (this.includeAnnotationConfig) {

AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
    }
    return this.registry.getBeanDefinitionCount() -
beanCountAtScanStart;
}
```

scan是个全局方法，扫描工作通过doScan(basePackages)委托给了doScan方法，同时，还包括了 includeAnnotationConfig属性的处理，AnnotationConfigUtils.registerAnnotation ConfigProcessors (this.registry)代

码主要是完成对于注解处理器的简单注册，比如
AutowiredAnnotationBeanPost Processor、
RequiredAnnotationBeanPostProcessor等，这里不再赘述，我们重点研究文件扫描功能的实现。

```
ClassPathMapperScanner.java
public Set<BeanDefinitionHolder> doScan(String... basePackages) {
    Set<BeanDefinitionHolder> beanDefinitions =
super.doScan(basePackages);
if (beanDefinitions.isEmpty()) {
//如果没有扫描到任何文件发出警告
    logger.warn("No MyBatis mapper was found in '" +
Arrays.toString(basePackages) + "'
    package. Please check your configuration.");
} else {
    for (BeanDefinitionHolder holder : beanDefinitions) {
        GenericBeanDefinition definition = (GenericBeanDefinition)
holder.getBeanDefinition();
        if (logger.isDebugEnabled()) {
            logger.debug("Creating MapperFactoryBean with name '" +
holder.getBeanName()
            + "' and '" + definition.getBeanClassName() + "'
mapperInterface");
        }
          //开始构造MapperFactoryBean类型的bean.
        definition.getPropertyValues().add("mapperInterface",
definition.getBeanClassName());
        definition.setBeanClass(MapperFactoryBean.class);
```

```
                    definition.getPropertyValues().add("addToConfig",
this.addToConfig);
                    boolean explicitFactoryUsed = false;
                    if (StringUtils.hasText(this.sqlSessionFactoryBeanName)) {
                        definition.getPropertyValues().add("sqlSessionFactory", new
RuntimeBeanReference
                       (this.sqlSessionFactoryBeanName));
                        explicitFactoryUsed = true;
                    } else if (this.sqlSessionFactory != null) {
                        definition.getPropertyValues().add("sqlSessionFactory",
this.sqlSessionFactory);
                        explicitFactoryUsed = true;
                   }
                    if (StringUtils.hasText(this.sqlSessionTemplateBeanName)) {
                        if (explicitFactoryUsed) {
                            logger.warn("Cannot use both: sqlSessionTemplate and
sqlSessionFactory together.
                            sqlSessionFactory is ignored.");
                        }
                        definition.getPropertyValues().add("sqlSessionTemplate", new
RuntimeBeanReference
                       (this.sqlSessionTemplateBeanName));
                        explicitFactoryUsed = true;
                    } else if (this.sqlSessionTemplate != null) {
                        if (explicitFactoryUsed) {
                            logger.warn("Cannot use both: sqlSessionTemplate and
sqlSessionFactory together.
```

```
                    sqlSessionFactory is ignored.");
                }
                definition.getPropertyValues().add("sqlSessionTemplate",
this.sqlSessionTemplate);
                explicitFactoryUsed = true;
            }
            if (!explicitFactoryUsed) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Enabling autowire by type for
MapperFactoryBean with name '" +
                    holder.getBeanName() + "'.");
                }

definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYP
            }
        }
    }
```

此时，虽然还没有完成介绍到扫描的过程，但是我们也应该理解了Spring中对于自动扫描的注册，声明 MapperScannerConfigurer 类型的 bean 目的是不需要我们对于每个接口都注册一个 MapperFactoryBean 类型的对应的 bean，但是，不在配置文件中注册并不代表这个 bean不存在，而是在扫描的过程中通过编码的方式动态注册。实现过程我们在上面的函数中可以看得非常清楚。

```
    protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
        Assert.notEmpty(basePackages, "At least one base package must be
specified");
        Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet
```

```
<BeanDefinition
        Holder>();
        for (String basePackage : basePackages) {
          //扫描basePackage路径下java文件
          Set<BeanDefinition> candidates =
findCandidateComponents(basePackage);
          for (BeanDefinition candidate : candidates) {
            //解析scope属性
            ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata
            (candidate);
            candidate.setScope(scopeMetadata.getScopeName());
            String beanName =
this.beanNameGenerator.generateBeanName(candidate,
            this.registry);
            if (candidate instanceof AbstractBeanDefinition) {
              postProcessBeanDefinition((AbstractBeanDefinition)
candidate,
              beanName);
            }
            if (candidate instanceof AnnotatedBeanDefinition) {
              //如果是 AnnotatedBeanDefinition 类型的 bean,需要检测下
常用注解如：
              Primary、Lazy等
              AnnotationConfigUtils.processCommonDefinitionAnnotations
              ((AnnotatedBeanDefinition) candidate);
            }
```

```
                //检测当前bean是否已经注册
                if (checkCandidate(beanName, candidate)) {
                    BeanDefinitionHolder definitionHolder = new
BeanDefinitionHolder
                    (candidate, beanName);
                   //如果当前bean是用于生成代理的bean那么需要进一步处
理
                    definitionHolder =

AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata,
definitionHolder, this.registry);
                   beanDefinitions.add(definitionHolder);
                    registerBeanDefinition(definitionHolder, this.registry);
                }
            }
        }
        return beanDefinitions;
    }
    public Set<BeanDefinition> findCandidateComponents(String
basePackage) {
        Set<BeanDefinition> candidates = new
LinkedHashSet<BeanDefinition>();
    try {
        String packageSearchPath = ResourcePatternResolver.
CLASSPATH_ ALL_URL_
        PREFIX +
            resolveBasePackage(basePackage) + "/" + this.resourcePattern;
```

```
Resource[] resources = this.resourcePatternResolver.getResources
(package
SearchPath);
boolean traceEnabled = logger.isTraceEnabled();
boolean debugEnabled = logger.isDebugEnabled();
for (Resource resource : resources) {
    if (traceEnabled) {
        logger.trace("Scanning " + resource);
    }
    if (resource.isReadable()) {
        try {
            MetadataReader metadataReader =
this.metadataReaderFactory.
            getMetadataReader(resource);
            if (isCandidateComponent(metadataReader)) {
                ScannedGenericBeanDefinition sbd = new
ScannedGenericBean
                Definition(metadataReader);
                sbd.setResource(resource);
                sbd.setSource(resource);
                if (isCandidateComponent(sbd)) {
                    if (debugEnabled) {
                        logger.debug("Identified candidate component
                        class: " + resource);
                    }
                    candidates.add(sbd);
                }
```

```
                    else {
                        if (debugEnabled) {
                            logger.debug("Ignored because not a concrete
                            top-level class: " + resource);
                        }
                    }
                }
                else {
                    if (traceEnabled) {
                        logger.trace("Ignored because not matching any
                        filter: " + resource);
                    }
                }
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                "Failed to read candidate component class: " + resource, ex);
            }
        }
        else {
            if (traceEnabled) {
                logger.trace("Ignored because not readable: " + resource);
            }
        }
    }
}
    catch (IOException ex) {
```

throw new BeanDefinitionStoreException("I/O failure during
classpath
scanning", ex);
}
return candidates;
}

findCandidateComponents 方法根据传入的包路径信息并结合类文件路径拼接成文件的绝对路径，同时完成了文件的扫描过程并且根据对应的文件生成了对应的 bean ，使用ScannedGenericBeanDefinition类型的bean承载信息，bean中只记录了resource和source信息。这里，我们更感兴趣的是isCandidateComponent(metadataReader)，此句代码用于判断当前扫描的文件是否符合要求，而我们之前注册的一些过滤器信息也正是在此时派上用场的。

protected boolean isCandidateComponent(MetadataReader
metadataReader) throws IOException {
for (TypeFilter tf : this.excludeFilters) {
if (tf.match(metadataReader, this.metadataReaderFactory)) {
return false;
}
}
for (TypeFilter tf : this.includeFilters) {
if (tf.match(metadataReader, this.metadataReaderFactory)) {
AnnotationMetadata metadata =
metadataReader.getAnnotationMetadata();
if (!metadata.isAnnotated(Profile.class.getName())) {
return true;
}

```
                AnnotationAttributes profile =
MetadataUtils.attributesFor(metadata,
               Profile.class);
                return
this.environment.acceptsProfiles(profile.getStringArray("value"));
            }
        }
        return false;
    }
```

我们看到了之前加入过滤器的两个属性 excludeFilters、includeFilters，并且知道对应的文件是否符合要求是根据过滤器中的match方法所返回的信息来判断的，当然用户可以实现并注册满足自己业务逻辑的过滤器来控制扫描的结果，metadataReader 中有你过滤所需要的全部文件信息。至此，我们完成了文件的扫描过程的分析。

第10章　事务

Spring声明式事务让我们从复杂的事务处理中得到解脱，使我们再也不需要去处理获得连接、关闭连接、事务提交和回滚等操作，再也不需要在与事务相关的方法中处理大量的try…catch…finally代码。Spring中事务的使用虽然已经相对简单得多，但是，还是有很多的使用及配置规则，有兴趣的读者可以自己查阅相关资料进行深入研究，这里只列举出最常用的使用方法。

同样，我们还是以最简单的示例来进行直观地介绍。

10.1 JDBC方式下的事务使用示例

（1）创建数据表结构。

```
CREATE TABLE 'user' (
    'id' int(11) NOT NULL auto_increment,
    'name' varchar(255) default NULL,
    'age' int(11) default NULL,
    'sex' varchar(255) default NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

（2）创建对应数据表的PO。

```
public class User {
    private int id;
    private String name;
    private int age;
private String sex;
//省略set/get方法
}
```

（3）创建表与实体间的映射。

```
public class UserRowMapper implements RowMapper {
   @Override
    public Object mapRow(ResultSet set, int index) throws
SQLException {
        User person = new User(set.getInt("id"), set.getString("name"), set
        .getInt("age"), set.getString("sex"));
        return person;
    }
}
```

（4）创建数据操作接口。

```
@Transactional(propagation=Propagation.REQUIRED)
public interface UserService {
    public void save(User user) throws Exception;
}
```

（5）创建数据操作接口实现类。

```
public class UserServiceImpl implements UserService {
    private JdbcTemplate jdbcTemplate;
   // 设置数据源
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
    public void save(User user) throws Exception {
        jdbcTemplate.update("insert into user(name,age,sex)values(?,?,?)",
        new Object[] { user.getName(), user.getAge(),
        user.getSex() }, new int[] { java.sql.Types.VARCHAR,
        java.sql.Types.INTEGER, java.sql.Types.VARCHAR });
       //事务测试，加上这句代码则数据不会保存到数据库中
        throw new RuntimeException("aa");
    }
}
```

（6）创建Spring配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:tx="http://www.Springframework.org/schema/tx"
   xmlns:context="http://www.Springframework.org/schema/context"
   xsi:schemaLocation="
```

```
                http://www.Springframework.org/schema/beans
http://www.Springframework.
               org/schema/beans/Spring-beans-2.5.xsd
            http://www.Springframework.org/schema/context
http://www.Springframework.
           org/schema/context/Spring-context-2.5.xsd
                http://www.Springframework.org/schema/tx
http://www.Springframework.org/
               schema/tx/Spring-tx-2.5.xsd
           ">
               <tx:annotation-driven transaction-manager="transactionManager"
/>
                <bean id="transactionManager"

class="org.Springframework.jdbc.datasource.DataSourceTransactionManager
                <property name="dataSource" ref="dataSource" />
           </bean>
            <!--配置数据源 -->
            <bean id="dataSource"
class="org.apache.commons.dbcp.BasicDataSource"
               destroy-method="close">
                <property name="driverClassName"
value="com.mysql.jdbc.Driver" />
                <property name="url"
value="jdbc:mysql://localhost:3306/lexueba" />
                <property name="username" value="root" />
                <property name="password" value="haojia0421xixi" />
```

```
            <!--连接池启动时的初始值 -->
            <property name="initialSize" value="1" />
            <!--连接池的最大值 -->
            <property name="maxActive" value="300" />
           <!-- 最大空闲值.当经过一个高峰时间后，连接池可以慢慢将已
经用不到的连接慢慢释放一部分，一直减
           少到maxIdle为止 -->
            <property name="maxIdle" value="2" />
            <!--最小空闲值.当空闲的连接数少于阀值时，连接池就会预申
请去一些连接，以免洪峰来时来不及申请 -->
            <property name="minIdle" value="1" />
        </bean>
         <!--配置业务bean：PersonServiceBean -->
         <bean id="userService" class="service.UserServiceImpl">
            <!--向属性dataSource注入数据源 -->
            <property name="dataSource" ref="dataSource"></property>
        </bean>
    </beans>
```

（7）测试。

```
    public static void main(String[] args) throws Exception {
         ApplicationContext act = new
ClassPathXmlApplicationContext("bean.xml");
         UserService userService = (UserService) act.getBean("userService");
         User user = new User();
        user.setName("张三ccc");
        user.setAge(20);
        user.setSex("男");
```

```
        // 保存一条记录
        userService.save(user);
        }
    }
```

上面的测试示例中，UserServiceImpl类对接口UserService中的save函数的实现最后加入了一句抛出异常的代码：throw new RuntimeException("aa")。当注掉这段代码执行测试类，那么会看到数据被成功的保存到了数据库中，但是如果加入这段代码时再次运行测试类，发现此处的操作并不会将数据保存到数据库中。

注意 默认情况下Spring中的事务处理只对RuntimeException方法进行回滚，所以，如果此处将RuntimeException替换成普通的Exception不会产生回滚效果。

10.2 事务自定义标签

对于Spring中事务功能的代码分析，我们首先从配置文件开始入手，在配置文件中有这样一个配置：<tx:annotation-driven />。可以说此处配置是事务的开关，如果没有此处配置，那么Spring中将不存在事务的功能。那么我们就从这个配置开始分析。

根据之前的分析，我们因此可以判断，在自定义标签中的解析过程中一定是做了一些辅助操作，于是我们先从自定义标签入手进行分析。

使用Eclipse搜索全局代码，关键字annotation-drive，最终锁定类TxNamespaceHandler，在TxNamespaceHandler中的init方法中：

```
public void init() {
    registerBeanDefinitionParser("advice", new TxAdviceBeanDefinitionParser());
    registerBeanDefinitionParser("annotation-driven", new
```

AnnotationDrivenBean

DefinitionParser());

registerBeanDefinitionParser("jta-transaction-manager", new JtaTransactionManagerBean

DefinitionParser());

}

根据自定义标签的使用规则以及上面的代码，可以知道，在遇到诸如<tx:annotation-driven为开头的配置后，Spring都会使用AnnotationDrivenBeanDefinitionParser类的parse方法进行解析。

```
public BeanDefinition parse(Element element, ParserContext parserContext) {
    String mode = element.getAttribute("mode");
    if ("aspectj".equals(mode)) {
        // mode="aspectj"
        registerTransactionAspect(element, parserContext);
    }else {
        // mode="proxy"
        AopAutoProxyConfigurer.configureAutoProxyCreator(element, parserContext);
    }
    return null;
}
```

在解析中存在对于 mode 属性的判断，根据代码，如果我们需要使用 AspectJ 的方式进行事务切入（Spring中的事务是以AOP为基础的），那么可以使用这样的配置：

```
<tx:annotation-driven transaction-manager="transactionManager" mode="aspectj" />
```

10.2.1 注册 InfrastructureAdvisorAutoProxyCreator

我们以默认配置为例子进行分析，进入AopAutoProxyConfigurer类的configureAutoProxyCreator：

```
public static void configureAutoProxyCreator(Element element,
ParserContext parserContext) {

AopNamespaceUtils.registerAutoProxyCreatorIfNecessary(parserContext,
element);
	//TRANSACTION_ADVISOR_BEAN_NAME
="org.Springframework.transaction.config.internal
	TransactionAdvisor";
	    String txAdvisorBeanName =
TransactionManagementConfigUtils.TRANSACTION_
	    ADVISOR_BEAN_NAME;
	    if (!parserContext.getRegistry().containsBeanDefinition
(txAdvisorBean Name)) {
	        Object eleSource = parserContext.extractSource(element);
	        //创建TransactionAttributeSource的bean
	        RootBeanDefinition sourceDef = new RootBeanDefinition
(Annotation
	        TransactionAttributeSource.class);
	        sourceDef.setSource(eleSource);
	        sourceDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
	    //注册bean,并使用Spring中的定义规则生成beanname
	String sourceName = parserContext.getReaderContext().
registerWithGeneratedName
```

```
(sourceDef);
  //创建TransactionInterceptor的bean
   RootBeanDefinition interceptorDef = new RootBeanDefinition
  (TransactionInterceptor.class);
  interceptorDef.setSource(eleSource);

interceptorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
   registerTransactionManager(element, interceptorDef);
  interceptorDef.getPropertyValues().add("transactionAttributeSource",
   new RuntimeBeanReference(sourceName));
  //注册bean,并使用Spring中的定义规则生成beanname
     String interceptorName = parserContext.getReaderContext().
Register
    WithGeneratedName(interceptorDef);
    //创建TransactionAttributeSourceAdvisor的bean
     RootBeanDefinition advisorDef = new RootBeanDefinition
(BeanFactory
    TransactionAttributeSourceAdvisor.class);
    advisorDef.setSource(eleSource);
    advisorDef.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
  //将sourceName的bean注入advisorDef的transactionAttributeSource
属性中
    advisorDef.getPropertyValues().add("transactionAttributeSource",
     new RuntimeBeanReference(sourceName));
  //将interceptorName的bean注入advisorDef的adviceBeanName属性
中
    advisorDef.getPropertyValues().add("adviceBeanName",
```

```
interceptorName);
                //如果配置了order属性，则加入到bean中
                 if (element.hasAttribute("order")) {
                     advisorDef.getPropertyValues().add("order",
element.getAttribute
                    ("order"));
                }

parserContext.getRegistry().registerBeanDefinition(txAdvisorBeanName,
                advisorDef);
                //创建CompositeComponentDefinition
                 CompositeComponentDefinition compositeDef = new
CompositeComponent
                 Definition(element.getTagName(), eleSource);
                 compositeDef.addNestedComponent(new
BeanComponentDefinition (sourceDef,
                sourceName));
             compositeDef.addNestedComponent(new
BeanComponentDefinition(interceptorDef,
            interceptorName));
                 compositeDef.addNestedComponent(new
BeanComponentDefinition(advisorDef,
                txAdvisorBeanName));
                parserContext.registerComponent(compositeDef);
            }
        }
```
上面的代码注册了代理类及三个bean，很多读者会直接略过，认为

只是注册三个bean 而已，确实，这里只注册了三个bean，但是这三个bean支撑了整个的事务功能，那么这三个bean是怎么组织起来的呢？

首先，其中的两个 bean 被注册到了一个名为 advisorDef 的 bean 中，advisorDef 使用BeanFactoryTransactionAttributeSourceAdvisor 作为其class 属性。也就是说BeanFactoryTransaction AttributeSourceAdvisor代表着当前bean，如图10-1所示，具体代码如下：

advisorDef.getPropertyValues().add("adviceBeanName", interceptorName);

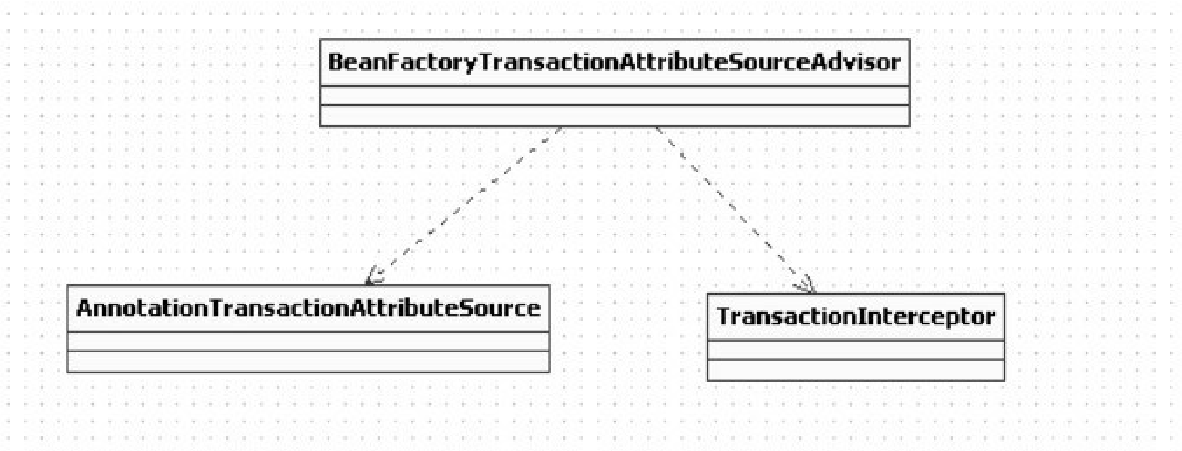


图10-1 BeanFactoryTransactionAttributeSourceAdvisor的组装

那么如此组装的目的是什么呢？我们暂且留下一个悬念，接着分析代码。上面函数configureAutoProxyCreator中的第一句貌似很简单但却是很重要的代码：

AopNamespaceUtils.registerAutoProxyCreatorIfNecessary(parserContex element);

进入这个函数：

public static void registerAutoProxyCreatorIfNecessary(
ParserContext parserContext, Element sourceElement) {
BeanDefinition beanDefinition =

```
AopConfigUtils.registerAutoProxyCreatorIf
            Necessary(
                 parserContext.getRegistry(), parserContext.extractSource (source
                Element));
             useClassProxyingIfNecessary(parserContext.getRegistry(),
sourceElement);
             registerComponentIfNecessary(beanDefinition, parserContext);
        }
        public static BeanDefinition
registerAutoProxyCreatorIfNecessary(BeanDefinitionRegistry
        registry, Object source) {
             return
registerOrEscalateApcAsRequired(InfrastructureAdvisorAutoProxyCreator.
             class, registry, source);
        }
```

对于解析来的代码流程 AOP 中已经有所分析，上面的两个函数主要目的是注册了InfrastructureAdvisorAutoProxyCreator 类型的 bean，那么注册这个类的目的是什么呢？查看这个类的层次，如图10-2所示。

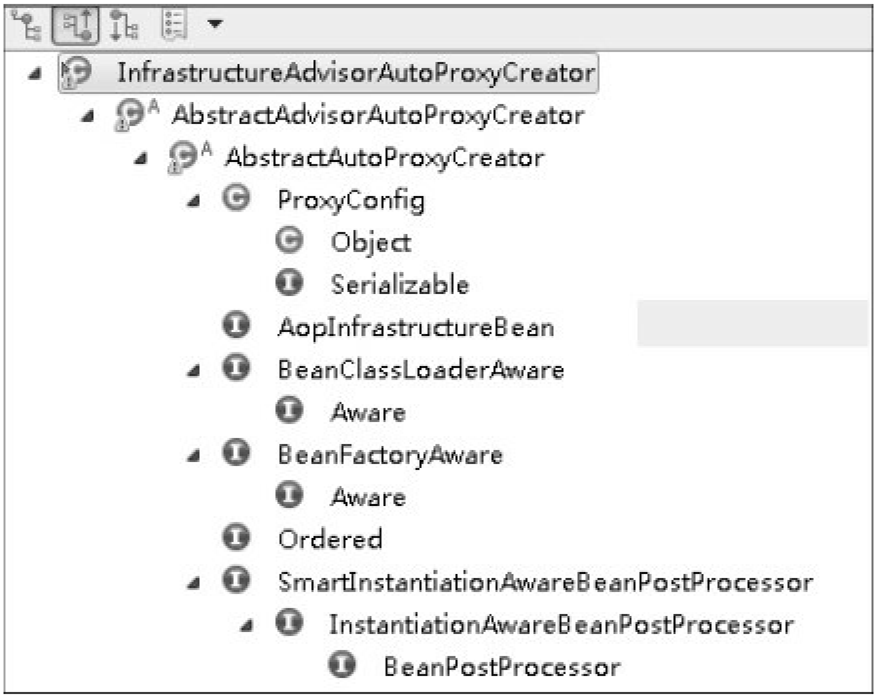


图10-2 InfrastructureAdvisorAutoProxyCreator类的层次结构图

从上面的层次结构中可以看到，InfrastructureAdvisorAutoProxyCreator 间接实现了SmartInstantiationAwareBeanPostProcessor，而SmartInstantiationAwareBeanPostProcessor又继承自InstantiationAwareBeanPostProcessor，也就是说在Spring中，所有bean实例化时Spring都会保证调用其postProcessAfterInitialization方法，其实现是在父类AbstractAutoProxyCreator类中实现。

以之前的示例为例，当实例化userService的bean时便会调用此方法，方法如下：

```
public Object postProcessAfterInitialization(Object bean, String beanName) throws
    BeansException {
        if (bean != null) {
```

//根据给定的bean的class和name构建出个key，beanClassName_beanName

Object cacheKey = getCacheKey(bean.getClass(), beanName);

//是否是由于避免循环依赖而创建的bean代理

if (!this.earlyProxyReferences.contains(cacheKey)) {

return wrapIfNecessary(bean, beanName, cacheKey);

}

}

return bean;

}

这里实现的主要目的是对指定 bean 进行封装，当然首先要确定是否需要封装，检测及封装的工作都委托给了wrapIfNecessary函数进行。

protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {

//如果已经 处理过

if (this.targetSourcedBeans.contains(beanName)) {

return bean;

}

if (this.nonAdvisedBeans.contains(cacheKey)) {

return bean;

}

//给定的bean类是否代表一个基础设施类，不应代理,或者配置了指定bean不需要自动代理

if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(),

beanName)) {

this.nonAdvisedBeans.add(cacheKey);

```
            return bean;
        }
        // Create proxy if we have advice.
        Object[] specificInterceptors = 
getAdvicesAndAdvisorsForBean(bean.getClass(),
        beanName, null);
        if (specificInterceptors != DO_NOT_PROXY) {
            this.advisedBeans.add(cacheKey);
            Object proxy = createProxy(bean.getClass(), beanName, 
specificInterceptors,
            new SingletonTargetSource(bean));
            this.proxyTypes.put(cacheKey, proxy.getClass());
            return proxy;
        }
        this.nonAdvisedBeans.add(cacheKey);
        return bean;
    }
```

wrapIfNecessary 函数功能实现起来很复杂，但是逻辑上理解起来还是相对简单的，在wrapIfNecessary函数中主要的工作如下。

（1）找出指定bean对应的增强器。

（2）根据找出的增强器创建代理。

听起来似乎简单的逻辑，Spring中又做了哪些复杂的工作呢？对于创建代理的部分，通过之前的分析相信大家已经很熟悉了，但是对于增强器的获取，Spring又是怎么做的呢？

10.2.2 获取对应class/method的增强器

获取指定 bean 对应的增强器，其中包含两个关键字：增强器与对

应。也就是说在getAdvicesAndAdvisorsForBean 函数中，不但要找出增强器，而且还需要判断增强器是否满足要求。

```
protected Object[] getAdvicesAndAdvisorsForBean(Class beanClass,
String beanName,
        TargetSource targetSource) {
    List advisors = findEligibleAdvisors(beanClass, beanName);
    if (advisors.isEmpty()) {
        return DO_NOT_PROXY;
    }
    return advisors.toArray();
}
protected List<Advisor> findEligibleAdvisors(Class beanClass, String
beanName) {
    List<Advisor> candidateAdvisors = findCandidateAdvisors();
    List<Advisor> eligibleAdvisors =
findAdvisorsThatCanApply(candidateAdvisors,
    beanClass, beanName);
    extendAdvisors(eligibleAdvisors);
    if (!eligibleAdvisors.isEmpty()) {
        eligibleAdvisors = sortAdvisors(eligibleAdvisors);
    }
    return eligibleAdvisors;
}
```

其实我们也渐渐地体会到了 Spring中代码的优秀，即使是一个很复杂的逻辑，在 Spring中也会被拆分成若干个小的逻辑，然后在每个函数中实现，使得每个函数的逻辑简单到我们能快速地理解，而不会像有些人开发的那样，将一大堆的逻辑都罗列在一个函数中，给后期维护人员

造成巨大的困扰。

同样，通过上面的函数，Spring又将任务进行了拆分，分成了获取所有增强器与增强器是否匹配两个功能点。

1．寻找候选增强器

在findCandidateAdvisors函数中完成的就是获取增强器的功能。

```
protected List<Advisor> findCandidateAdvisors() {
    return this.advisorRetrievalHelper.findAdvisorBeans();
}
public List<Advisor> findAdvisorBeans() {
    // Determine list of advisor bean names, if not cached already.
    String[] advisorNames = null;
    synchronized (this) {
        advisorNames = this.cachedAdvisorBeanNames;
        if (advisorNames == null) {
            advisorNames =
BeanFactoryUtils.beanNamesForTypeIncludingAncestors(
                this.beanFactory, Advisor.class, true, false);
                this.cachedAdvisorBeanNames = advisorNames;
        }
    }
    if (advisorNames.length == 0) {
        return new LinkedList<Advisor>();
    }
    List<Advisor> advisors = new LinkedList<Advisor>();
    for (String name : advisorNames) {
        if (isEligibleBean(name) &&
!this.beanFactory.isCurrentlyInCreation(name)) {
```

```
                    try {
                        advisors.add(this.beanFactory.getBean(name, Advisor.class));
                    }
                    catch (BeanCreationException ex) {
                        Throwable rootCause = ex.getMostSpecificCause();
                        if (rootCause instanceof BeanCurrentlyInCreationException)
{
                            BeanCreationException bce = (BeanCreationException)
rootCause;
                            if
(this.beanFactory.isCurrentlyInCreation(bce.getBeanName())) {
                                if (logger.isDebugEnabled()) {
                                    logger.debug("Ignoring currently created advisor '"
                                    + name + "': " + ex.getMessage());
                                }
                                continue;
                            }
                        }
                        throw ex;
                    }
                }
            }
            return advisors;
        }
```

对于上面的函数，你看懂其中的奥妙了吗？首先是通过BeanFactoryUtils 类提供的工具方法获取所有对应Advisor.class的类，获取办法无非是使用ListableBeanFactory中提供的方法：

```
String[] getBeanNamesForType(Class<?> type, boolean
includeNonSingletons, boolean
allowEagerInit);
```

而当我们知道增强器在容器中的beanName时，获取增强器已经不是问题了，在BeanFactory中提供了这样的方法，可以帮助我们快速定位对应的bean实例。

```
<T> T getBean(String name, Class<T> requiredType) throws
BeansException;
```

或许你已经忘了之前留下的悬念，在我们讲解自定义标签时曾经注册了一个类型为BeanFactoryTransactionAttributeSourceAdvisor 的 bean，而在此 bean 中我们又注入了另外两个Bean，那么此时这个 Bean 就会被开始使用了。因为 BeanFactoryTransactionAttribute Source Advisor同样也实现了Advisor接口，那么在获取所有增强器时自然也会将此bean提取出来，并随着其他增强器一起在后续的步骤中被织入代理。

2．候选增强器中寻找到匹配项

当找出对应的增强器后，接来的任务就是看这些增强器是否与对应的 class 匹配了，当然不只是class，class内部的方法如果匹配也可以通过验证。

```
public static List<Advisor> findAdvisorsThatCanApply(List<Advisor>
candidateAdvisors,
Class<?> clazz) {
    if (candidateAdvisors.isEmpty()) {
        return candidateAdvisors;
    }
    List<Advisor> eligibleAdvisors = new LinkedList<Advisor>();
    //首先处理引介增强
    for (Advisor candidate : candidateAdvisors) {
```

```
            if (candidate instanceof IntroductionAdvisor &&
canApply(candidate, clazz)) {
                eligibleAdvisors.add(candidate);
            }
        }
        boolean hasIntroductions = !eligibleAdvisors.isEmpty();
        for (Advisor candidate : candidateAdvisors) {
            //引介增强已经处理
            if (candidate instanceof IntroductionAdvisor) {
                continue;
            }
            //对于普通bean的处理
            if (canApply(candidate, clazz, hasIntroductions)) {
                eligibleAdvisors.add(candidate);
            }
        }
        return eligibleAdvisors;
    }
    public static boolean canApply(Advisor advisor, Class<?> targetClass,
boolean
    hasIntroductions) {
        if (advisor instanceof IntroductionAdvisor) {
            return((IntroductionAdvisor)advisor).getClassFilter().Matches
            (targetClass);
        }else if (advisor instanceof PointcutAdvisor) {
            PointcutAdvisor pca = (PointcutAdvisor) advisor;
            return canApply(pca.getPointcut(), targetClass, hasIntroductions);
```

```
        }else {
            return true;
        }
    }
```

当前我们分析的是对于UserService是否适用于此增强方法，那么当前的advisor就是之前查找出来的类型为BeanFactoryTransactionAttributeSourceAdvisor的bean实例，而通过类的层次结构我们又知道：BeanFactoryTransactionAttributeSourceAdvisor 间接实现了 PointcutAdvisor。因此，在 canApply 函数中的第二个 if 判断时就会通过判断，会将 BeanFactory Transaction AttributeSourceAdvisor 中的 getPointcut()方法返回值作为参数继续调用 canApply 方法，而 getPoint()方法返回的是 TransactionAttributeSourcePointcut 类型的实例。对于 transactionAttribute Source这个属性大家还有印象吗？这是在解析自定义标签时注入进去的。

```
    private final TransactionAttributeSourcePointcut pointcut = new
TransactionAttribute
    SourcePointcut() {
        @Override
        protected TransactionAttributeSource
getTransactionAttributeSource() {
            return transactionAttributeSource;
        }
    };
```

那么，使用ransactionAttributeSourcePointcut类型的实例作为函数参数继续跟踪canApply。

```
    public static boolean canApply(Pointcut pc, Class<?> targetClass,
boolean hasIntroductions) {
```

```
            Assert.notNull(pc, "Pointcut must not be null");
            if (!pc.getClassFilter().matches(targetClass)) {
                return false;
            }
            //此时的pc表示 TransactionAttributeSourcePointcut
            //pc.getMethodMatcher()返回的正是自身(this).
            MethodMatcher methodMatcher = pc.getMethodMatcher();
            IntroductionAwareMethodMatcher
introductionAwareMethodMatcher = null;
            if (methodMatcher instanceof IntroductionAwareMethodMatcher) {
                introductionAwareMethodMatcher =
(IntroductionAwareMethodMatcher) methodMatcher;
            }
            Set<Class> classes = new HashSet<Class>(ClassUtils.
GetAllInterfaces
            ForClassAsSet(targetClass));
            classes.add(targetClass);
            //classes:[interface test.IITestBean, class test.TestBean]
            for (Class<?> clazz : classes) {
                Method[] methods = clazz.getMethods();
                for (Method method : methods) {
                    if ((introductionAwareMethodMatcher != null &&
                        introductionAwareMethodMatcher.matches(method,
targetClass,
                        hasIntroductions)) ||
                       methodMatcher.matches(method, targetClass)) {
                        return true;
```

```
            }
        }
    }
    return false;
}
```

通过上面函数大致可以理清大体脉络，首先获取对应类的所有接口并连同类本身一起遍历，遍历过程中又对类中的方法再次遍历，一旦匹配成功便认为这个类适用于当前增强器。

到这里我们不禁会有疑问，对于事物的配置不仅仅局限于在函数上配置，我们都知道，在类活接口上的配置可以延续到类中的每个函数，那么，如果针对每个函数进行检测，在类本身上配置的事务属性岂不是检测不到了吗？带着这个疑问，我们继续探求matcher方法。

做匹配的时候methodMatcher.matches(method, targetClass)会使用TransactionAttributeSource Pointcut类的matches方法。

```
public boolean matches(Method method, Class targetClass) {
  //自定义标签解析时注入
    TransactionAttributeSource tas = getTransactionAttributeSource();
    return (tas == null || tas.getTransactionAttribute(method, targetClass)
!= null);
}
```

此时的tas表示AnnotationTransactionAttributeSource类型，而AnnotationTransactionAttribute Source类型的getTransactionAttribute方法如下：

```
public TransactionAttribute getTransactionAttribute(Method method,
Class<?>
targetClass) {
    Object cacheKey = getCacheKey(method, targetClass);
```

```
            Object cached = this.attributeCache.get(cacheKey);
            if (cached != null) {
                if (cached == NULL_TRANSACTION_ATTRIBUTE) {
                    return null;
                }
                else {
                    return (TransactionAttribute) cached;
                }
            }
            else {
                TransactionAttribute txAtt =
computeTransactionAttribute(method, targetClass);
                // Put it in the cache.
                if (txAtt == null) {
                    this.attributeCache.put(cacheKey,
NULL_TRANSACTION_ATTRIBUTE);
                }
                else {
                    if (logger.isDebugEnabled()) {
                        logger.debug("Adding transactional method '" +
method.getName()
                        + "' with attribute: " + txAtt);
                    }
                    this.attributeCache.put(cacheKey, txAtt);
                }
                return txAtt;
            }
```

}

很遗憾，在getTransactionAttribute函数中并没有找到我们想要的代码，这里是指常规的一贯的套路。尝试从缓存加载，如果对应信息没有被缓存的话，工作又委托给了 computeTransaction Attribute函数，在computeTransactionAttribute函数中终于的我们看到了事务标签的提取过程。

3．提取事务标签

```
private TransactionAttribute computeTransactionAttribute(Method method, Class<?>
targetClass) {
    // Don't allow no-public methods as required.
    if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {
        return null;
    }
    // Ignore CGLIB subclasses - introspect the actual user class.
    Class<?> userClass = ClassUtils.getUserClass(targetClass);
    //method代表接口中的方法，specificMethod代表实现类中的方法
    Method specificMethod = ClassUtils.getMostSpecificMethod(method, userClass);
    // If we are dealing with method with generic parameters, find the original method.
    specificMethod = BridgeMethodResolver.findBridgedMethod(specificMethod);
    //查看方法中是否存在事务声明
    TransactionAttribute txAtt = findTransactionAttribute(specificMethod);
```

```
        if (txAtt != null) {
            return txAtt;
        }
        //查看方法所在类中是否存在事务声明
        txAtt =
findTransactionAttribute(specificMethod.getDeclaringClass());
        if (txAtt != null) {
            return txAtt;
        }
        //如果存在接口，则到接口中去寻找
        if (specificMethod != method) {
            //查找接口方法
            txAtt = findTransactionAttribute(method);
            if (txAtt != null) {
                return txAtt;
            }
            //到接口中的类中去寻找
            return findTransactionAttribute(method.getDeclaringClass());
        }
        return null;
    }
```

对于事务属性的获取规则相信大家都已经很清楚，如果方法中存在事务属性，则使用方法上的属性，否则使用方法所在的类上的属性，如果方法所在类的属性上还是没有搜寻到对应的事务属性，那么再搜寻接口中的方法，再没有的话，最后尝试搜寻接口的类上面的声明。对于函数computeTransactionAttribute中的逻辑与我们所认识的规则并无差别，但是上面函数中并没有真正的去做搜寻事务属性的逻辑，而是搭建了个

执行框架，将搜寻事务属性的任务委托给了findTransactionAttribute方法去执行。

```
protected TransactionAttribute findTransactionAttribute(Method method) {
    return determineTransactionAttribute(method);
}
protected TransactionAttribute determineTransactionAttribute(AnnotatedElement ae) {
     for (TransactionAnnotationParser annotationParser : this.annotationParsers) {
         TransactionAttribute attr = annotationParser.parseTransaction Annotation (ae);
         if (attr != null) {
             return attr;
         }
     }
     return null;
}
```

this.annotationParsers是在当前类AnnotationTransactionAttributeSource初始化的时候初始化的，其中的值被加入了SpringTransactionAnnotationParser，也就是当进行属性获取的时候其实是使用 SpringTransactionAnnotationParser 类的 parseTransactionAnnotation 方法进行解析的。

```
public TransactionAttribute parseTransactionAnnotation(AnnotatedElement ae) {
     Transactional ann = AnnotationUtils.getAnnotation(ae, Transactional.class);
```

```
    if (ann != null) {
        return parseTransactionAnnotation(ann);
    }
    else {
        return null;
    }
}
```

至此，我们终于看到了想看到的获取注解标记的代码。首先会判断当前的类是否含有Transactional 注解，这是事务属性的基础，当然如果有的话会继续调用 parseTransactionAnnotation方法解析详细的属性。

```
public TransactionAttribute parseTransactionAnnotation(Transactional
ann) {
    RuleBasedTransactionAttribute rbta = new
RuleBasedTransactionAttribute();
    //解析propagation
    rbta.setPropagationBehavior(ann.propagation().value());
    //解析isolation
    rbta.setIsolationLevel(ann.isolation().value());
    //解析timeout
    rbta.setTimeout(ann.timeout());
    //解析readOnly
    rbta.setReadOnly(ann.readOnly());
    //解析value
    rbta.setQualifier(ann.value());
    ArrayList<RollbackRuleAttribute> rollBackRules = new
ArrayList<RollbackRule
    Attribute>();
```

```
//解析rollbackFor
 Class[] rbf = ann.rollbackFor();
 for (Class rbRule : rbf) {
     RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);
    rollBackRules.add(rule);
}
//解析rollbackForClassName
 String[] rbfc = ann.rollbackForClassName();
 for (String rbRule : rbfc) {
     RollbackRuleAttribute rule = new RollbackRuleAttribute(rbRule);
    rollBackRules.add(rule);
}
//解析noRollbackFor
 Class[] nrbf = ann.noRollbackFor();
 for (Class rbRule : nrbf) {
     NoRollbackRuleAttribute rule = new
NoRollbackRuleAttribute(rbRule);
    rollBackRules.add(rule);
}
//解析noRollbackForClassName
 String[] nrbfc = ann.noRollbackForClassName();
 for (String rbRule : nrbfc) {
     NoRollbackRuleAttribute rule = new
NoRollbackRuleAttribute(rbRule);
    rollBackRules.add(rule);
}
rbta.getRollbackRules().addAll(rollBackRules);
```

```
        return rbta;
    }
```

上面方法中实现了对对应类或者方法的事务属性解析，你会在这个类中看到任何你常用或者不常用的属性提取。

至此，我们终于完成了事务标签的解析。我们是不是分析的太远了，似乎已经忘了从哪里开始了。再回顾一下，我们的现在的任务是找出某个增强器是否适合于对应的类，而是否匹配的关键则在于是否从指定的类或类中的方法中找到对应的事务属性，现在，我们以UserServiceImpl为例，已经在它的接口 UserService 中找到了事务属性，所以，它是与事务增强器匹配的，也就是它会被事务功能修饰。

至此，事务功能的初始化工作便结束了，当判断某个 bean 适用于事务增强时，也就是适用于增强器BeanFactoryTransactionAttributeSourceAdvisor，没错，还是这个类，所以说，在自定义标签解析时，注入的类成为了整个事务功能的基础。

BeanFactoryTransactionAttributeSourceAdvisor作为Advisor的实现类，自然要遵从Advisor的处理方式，当代理被调用时会调用这个类的增强方法，也就是此 bean 的 Advise，又因为在解析事务定义标签时我们把 TransactionInterceptor 类型的 bean 注入到了 BeanFactory TransactionAttributeSourceAdvisor 中，所以，在调用事务增强器增强的代理类时会首先执行TransactionInterceptor进行增强，同时，也就是在TransactionInterceptor类中的invoke方法中完成了整个事务的逻辑。

10.3 事务增强器

TransactionInterceptor 支撑着整个事务功能的架构，逻辑还是相对复杂的，那么现在我们切入正题来分析此拦截器是如何实现事务特性的。TransactionInterceptor类继承自MethodInterceptor，所以调用该类是

从其invoke方法开始的，首先预览下这个方法：

```
    public Object invoke(final MethodInvocation invocation) throws
Throwable {
        Class<?> targetClass = (invocation.getThis() != null ?
AopUtils.getTargetClass
        (invocation.getThis()) : null);
        //获取对应事务属性
        final TransactionAttribute txAttr =
        getTransactionAttributeSource().getTransactionAttribute (invocation.
        getMethod(), targetClass);
    //获取beanFactory中的transactionManager
    final PlatformTransactionManager tm =
determineTransactionManager(txAttr);
    //构造方法唯一标识（类.方法，如service.UserServiceImpl.save）
    final String joinpointIdentification = methodIdentification (invocation.
    getMethod(), targetClass);
        //声明式事务处理
        if (txAttr == null || !(tm instanceof
CallbackPreferringPlatformTransaction
        Manager)) {
          //创建TransactionInfo
          TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr,
joinpoint
          Identification);
          Object retVal = null;
          try {
            //执行被增强方法
```

```
            retVal = invocation.proceed();
        }
        catch (Throwable ex) {
            //异常回滚
            completeTransactionAfterThrowing(txInfo, ex);
            throw ex;
        }
        finally {
            //清除信息
            cleanupTransactionInfo(txInfo);
        }
        //提交事务
        commitTransactionAfterReturning(txInfo);
        return retVal;
    }
    else {
        //编程式事务处理
        try {
            Object result =
((CallbackPreferringPlatformTransactionManager) tm).
            execute(txAttr,
            new TransactionCallback<Object>() {
                public Object doInTransaction(TransactionStatus status) {
                    TransactionInfo txInfo = prepareTransactionInfo(tm,
                    txAttr, joinpointIdentification, status);
                        try {
                            return invocation.proceed();
```

```
            }
            catch (Throwable ex) {
                if (txAttr.rollbackOn(ex)) {
                    // A RuntimeException: will lead to a rollback.
                    if (ex instanceof RuntimeException) {
                        throw (RuntimeException) ex;
                    }
                    else {
                        throw new ThrowableHolderException(ex);
                    }
                }
                else {
                    // A normal return value: will lead to a commit.
                    return new ThrowableHolder(ex);
                }
            }
            finally {
                cleanupTransactionInfo(txInfo);
            }
        }
    });
// Check result: It might indicate a Throwable to rethrow.
if (result instanceof ThrowableHolder) {
    throw ((ThrowableHolder) result).getThrowable();
}
else {
    return result;
```

```
            }
        }
        catch (ThrowableHolderException ex) {
            throw ex.getCause();
        }
    }
}
```

从上面的函数中，我们尝试整理下事务处理的脉络，在Spring中支持两种事务处理的方式，分别是声明式事务处理与编程式事务处理，两者相对于开发人员来讲差别很大，但是对于Spring中的实现来讲，大同小异。在invoke中我们也可以看到这两种方式的实现。考虑到对事务的应用比声明式的事务处理使用起来方便，也相对流行些，我们就以此种方式进行分析。对于声明式的事务处理主要有以下几个步骤。

（1）获取事务的属性。

对于事务处理来说，最基础或者说最首要的工作便是获取事务属性了，这是支撑整个事务功能的基石，如果没有事务属性，其他功能也无从谈起，在分析事务准备阶段时我们已经分析了事务属性提取的功能，大家应该有所了解。

（2）加载配置中配置的TransactionManager。

（3）不同的事务处理方式使用不同的逻辑。

对于声明式事务的处理与编程式事务的处理，第一点区别在于事务属性上，因为编程式的事务处理是不需要有事务属性的，第二点区别就是在TransactionManager上，CallbackPreferring PlatformTransactionManager 实现 PlatformTransactionManager 接口，暴露出一个方法用于执行事务处理中的回调。所以，这两种方式都可以用作事务处理方式的判断。

（4）在目标方法执行前获取事务并收集事务信息。

事务信息与事务属性并不相同，也就是TransactionInfo与TransactionAttribute并不相同， TransactionInfo中包含TransactionAttribute信息，但是，除了TransactionAttribute外还有其他事务信息，例如PlatformTransactionManager以及TransactionStatus相关信息。

（5）执行目标方法。

（6）一旦出现异常，尝试异常处理。

并不是所有异常，Spring都会将其回滚，默认只对RuntimeException回滚。

（7）提交事务前的事务信息清除。

（8）提交事务。

上面的步骤分析旨在让大家对事务功能与步骤有个大致的了解，具体的功能还需要详细地分析。

10.3.1 创建事务

我们先分析事务创建的过程。

```
protected TransactionInfo createTransactionIfNecessary(
PlatformTransactionManager tm, TransactionAttribute txAttr, final
String
joinpointIdentification) {
    // If no name specified, apply method identification as transaction
name.
    //如果没有名称指定则使用方法唯一标识，并使用
DelegatingTransactionAttribute封装txAttr
    if (txAttr != null && txAttr.getName() == null) {
        txAttr = new DelegatingTransactionAttribute(txAttr) {
            @Override
```

```
            public String getName() {
                return joinpointIdentification;
            }
        };
    }
    TransactionStatus status = null;
    if (txAttr != null) {
        if (tm != null) {
           //获取TransactionStatus
            status = tm.getTransaction(txAttr);
        }
        else {
            if (logger.isDebugEnabled()) {
                logger.debug("Skipping transactional joinpoint [" + joinpoint
                Identification +
                "] because no transaction manager has been configured");
            }
        }
    }
   //根据指定的属性与status准备一个TransactionInfo
    return prepareTransactionInfo(tm, txAttr, joinpointIdentification,
status);
}
```

对于createTransactionIfNecessar函数主要做了这样几件事情。

（1）使用DelegatingTransactionAttribute封装传入的TransactionAttribute实例。

对于传入的TransactionAttribute类型的参数txAttr，当前的实际类型

是RuleBasedTransaction Attribute，是由获取事务属性时生成，主要用于数据承载，而这里之所以使用 Delegating TransactionAttribute进行封装，当然是提供了更多的功能。

（2）获取事务。

事务处理当然是以事务为核心，那么获取事务就是最重要的事情。

（3）构建事务信息。

根据之前几个步骤获取的信息构建TransactionInfo并返回。

我们分别对以上步骤进行详细的解析。

1．获取事务

Spring中使用getTransaction来处理事务的准备工作，包括事务获取以及信息的构建。

```
public final TransactionStatus getTransaction(TransactionDefinition definition) throws
TransactionException {
    Object transaction = doGetTransaction();
    // Cache debug flag to avoid repeated checks.
    boolean debugEnabled = logger.isDebugEnabled();
    if (definition == null) {
        // Use defaults if no transaction definition given.
        definition = new DefaultTransactionDefinition();
    }
//判断当前线程是否存在事务，判读依据为当前线程记录的连接不为空且连接中(connectionHolder)中的
transactionActive属性不为空
    if (isExistingTransaction(transaction)) {
      //当前线程已经存在事务
        return handleExistingTransaction(definition, transaction,
```

```
debugEnabled);
            }
            //事务超时设置验证
             if (definition.getTimeout() <
TransactionDefinition.TIMEOUT_DEFAULT) {
                 throw new InvalidTimeoutException("Invalid transaction timeout",
                definition.getTimeout());
            }
            //如果当前线程不存在事务，但是propagationBehavior却被声明为
PROPAGATION_MANDATORY抛
            出异常
             if (definition.getPropagationBehavior() == TransactionDefinition.
PROPAGATION_
            MANDATORY) {
                 throw new IllegalTransactionStateException(
                 "No existing transaction found for transaction marked with
                 propagation 'mandatory'");
             }else if (definition.getPropagationBehavior() ==
TransactionDefinition.
             PROPAGATION_REQUIRED ||
             definition.getPropagationBehavior() == TransactionDefinition.
PROPAGATION_
            REQUIRES_NEW ||
             definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NESTED) {
                //PROPAGATION_REQUIRED、
PROPAGATION_REQUIRES_NEW、PROPAGATION_NESTED都需要
```

新建事务

```
//空挂起
 SuspendedResourcesHolder suspendedResources = suspend(null);
 if (debugEnabled) {
     logger.debug("Creating new transaction with name [" +
definition.
     getName() + "]: " + definition);
}
 try {
     boolean newSynchronization =
(getTransactionSynchronization() !=
    SYNCHRONIZATION_NEVER);
     DefaultTransactionStatus status = newTransactionStatus(
     definition, transaction, true, newSynchronization,
debugEnabled,
    suspendedResources);
    /*
       * 构造transaction,包括设置ConnectionHolder、隔离级别、
timout
       * 如果是新连接，绑定到当前线程
       */
    doBegin(transaction, definition);
    //新同步事务的设置，针对于当前线程的设置
    prepareSynchronization(status, definition);
     return status;
}
 catch (RuntimeException ex) {
```

```
                resume(null, suspendedResources);
                throw ex;
            }
            catch (Error err) {
                resume(null, suspendedResources);
                throw err;
            }
        }
        else {
            // Create "empty" transaction: no actual transaction, but potentially
            synchronization.
            boolean newSynchronization = (getTransactionSynchronization()
==
            SYNCHRONIZATION_ALWAYS);
        return prepareTransactionStatus(definition, null, true,
newSynchronization,
        debugEnabled, null);
        }
    }
```

当然，在Spring中每个复杂的功能实现，并不是一次完成的，而是会通过入口函数进行一个框架的搭建，初步构建完整的逻辑，而将实现细节分摊给不同的函数。那么，让我们看看事务的准备工作都包括哪些。

（1）获取事务

创建对应的事务实例，这里使用的是DataSourceTransactionManager中的doGetTransaction方法，创建基于JDBC的事务实例。如果当前线程中存在关于dataSource的连接，那么直接使用。这里有一个对保存点的

设置，是否开启允许保存点取决于是否设置了允许嵌入式事务。

```
protected Object doGetTransaction() {
    DataSourceTransactionObject txObject = new
DataSourceTransactionObject();
    txObject.setSavepointAllowed(isNestedTransactionAllowed());
    //如果当前线程已经记录数据库连接则使用原有连接
    ConnectionHolder conHolder =
    (ConnectionHolder)
TransactionSynchronizationManager.getResource(this.dataSource);
      //false表示非新创建连接。
    txObject.setConnectionHolder(conHolder, false);
    return txObject;
}
```

（2）如果当先线程存在事务，则转向嵌套事务的处理。

（3）事务超时设置验证。

（4）事务propagationBehavior属性的设置验证。

（5）构建DefaultTransactionStatus。

（6）完善 transaction，包括设置 ConnectionHolder、隔离级别、timeout，如果是新连接，则绑定到当前线程。

对于一些隔离级别、timeout 等功能的设置并不是由 Spring 来完成的，而是委托给底层的数据库连接去做的，而对于数据库连接的设置就是在doBegin函数中处理的。

```
/**
  * 构造transaction,包括设置ConnectionHolder、隔离级别、timeout
  * 如果是新连接，绑定到当前线程
  */
@Override
```

```
protected void doBegin(Object transaction, TransactionDefinition
definition) {
    DataSourceTransactionObject txObject =
(DataSourceTransactionObject) transaction;
        Connection con = null;
        try {
            if (txObject.getConnectionHolder() == null ||

txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
                Connection newCon = this.dataSource.getConnection();
                if (logger.isDebugEnabled()) {
                    logger.debug("Acquired Connection [" + newCon + "] for JDBC
transaction");
                }
                txObject.setConnectionHolder(new ConnectionHolder(newCon),
true);
                }

txObject.getConnectionHolder().setSynchronizedWithTransaction(true);
                con = txObject.getConnectionHolder().getConnection();
                //设置隔离级别
                Integer previousIsolationLevel =
DataSourceUtils.prepareConnection
                ForTransaction(con, definition);
                txObject.setPreviousIsolationLevel(previousIsolationLevel);
                //更改自动提交设置，由Spring控制提交
                if (con.getAutoCommit()) {
```

```
                    txObject.setMustRestoreAutoCommit(true);
                     if (logger.isDebugEnabled()) {
                         logger.debug("Switching JDBC Connection [" + con + "] to
manual commit");
                    }
                    con.setAutoCommit(false);
                }
                //设置判断当前线程是否存在事务的依据
                txObject.getConnectionHolder().setTransactionActive(true);
                 int timeout = determineTimeout(definition);
                 if (timeout != TransactionDefinition.TIMEOUT_DEFAULT) {
                    txObject.getConnectionHolder().setTimeoutInSeconds(timeout);
                }
                 // Bind the session holder to the thread.
                 if (txObject.isNewConnectionHolder()) {
                    //将当前获取到的连接绑定到当前线程

TransactionSynchronizationManager.bindResource(getDataSource(),
                    txObject.getConnectionHolder());
                }
            }
             catch (Exception ex) {
                 DataSourceUtils.releaseConnection(con, this.dataSource);
                 throw new CannotCreateTransactionException("Could not open
JDBC Connection
                 for transaction", ex);
            }
```

}

可以说事务是从这个函数开始的，因为在这个函数中已经开始尝试了对数据库连接的获取，当然，在获取数据库连接的同时，一些必要的设置也是需要同步设置的。

n 尝试获取连接。

当然并不是每次都会获取新的连接，如果当前线程中的connectionHolder已经存在，则没有必要再次获取，或者，对于事务同步表示设置为true的需要重新获取连接。

o 设置隔离级别以及只读标识。

你是否有过这样的错觉？事务中的只读配置是Spring中做了一些处理呢？Spring中确实是针对只读操作做了一些处理，但是核心的实现是设置connection上的readOnly属性。同样，对于隔离级别的控制也是交由connection去控制的。

p 更改默认的提交设置。

如果事务属性是自动提交，那么需要改变这种设置，而将提交操作委托给Spring来处理。

q 设置标志位，标识当前连接已经被事务激活。

r 设置过期时间。

s 将connectionHolder绑定到当前线程。

设置隔离级别的prepareConnectionForTransaction函数用于负责对底层数据库连接的设置，当然，只是包含只读标识和隔离级别的设置。由于强大的日志及异常处理，显得函数代码量比较大，但是单从业务角度去看，关键代码其实是不多的。

public static Integer prepareConnectionForTransaction(Connection con, Transaction Definition

definition)

throws SQLException {

```
        Assert.notNull(con, "No Connection specified");
        //设置数据连接的只读标识
        if (definition != null && definition.isReadOnly()) {
            try {
                if (logger.isDebugEnabled()) {
                    logger.debug("Setting JDBC Connection [" + con + "] read-
only");
                }
                con.setReadOnly(true);
            }
            catch (SQLException ex) {
                Throwable exToCheck = ex;
                while (exToCheck != null) {
                    if
(exToCheck.getClass().getSimpleName().contains("Timeout")) {
                        // Assume it's a connection timeout that would otherwise get
                        lost: e.g. from JDBC 4.0
                        throw ex;
                    }
                    exToCheck = exToCheck.getCause();
                }
                logger.debug("Could not set JDBC Connection read-only", ex);
            }
            catch (RuntimeException ex) {
                Throwable exToCheck = ex;
                while (exToCheck != null) {
                    if
```

```
(exToCheck.getClass().getSimpleName().contains("Timeout")) {
                        throw ex;
                    }
                    exToCheck = exToCheck.getCause();
                }
                logger.debug("Could not set JDBC Connection read-only", ex);
            }
        }
        //设置数据库连接的隔离级别
        Integer previousIsolationLevel = null;
        if (definition != null && definition.getIsolationLevel() != Transaction
        Definition.ISOLATION_DEFAULT) {
            if (logger.isDebugEnabled()) {
                logger.debug("Changing isolation level of JDBC Connection [" +
con + "] to " +
                definition.getIsolationLevel());
                }
                int currentIsolation = con.getTransactionIsolation();
                if (currentIsolation != definition.getIsolationLevel()) {
                    previousIsolationLevel = currentIsolation;
                    con.setTransactionIsolation(definition.getIsolationLevel());
                }
            }
            return previousIsolationLevel;
        }
```

（7）将事务信息记录在当前线程中。

```
        protected void prepareSynchronization(DefaultTransactionStatus status,
```

```
Transaction
        Definition definition) {
            if (status.isNewSynchronization()) {

TransactionSynchronizationManager.setActualTransactionActive(status.
                hasTransaction());

TransactionSynchronizationManager.setCurrentTransactionIsolationLevel(
                 (definition.getIsolationLevel() !=
TransactionDefinition.ISOLATION_
                DEFAULT) ?
                 definition.getIsolationLevel() : null);

TransactionSynchronizationManager.setCurrentTransactionReadOnly
(definition.
                isReadOnly());

TransactionSynchronizationManager.setCurrentTransactionName(definition.
                getName());
                TransactionSynchronizationManager.initSynchronization();
            }
        }
```

2．处理已经存在的事务

之前讲述了普通事务建立的过程，但是Spring中支持多种事务的传播规则，比如PROPAGATION_NESTED、PROPAGATION_REQUIRES_NEW等，这些都是在已经存在事务的基础上进行进一步的处理，那么，对于已经存在的事务，准备操作是如何

进行的呢？

```
    private TransactionStatus handleExistingTransaction(
    TransactionDefinition definition, Object transaction, boolean
debugEnabled)
    throws TransactionException {
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NEVER) {
            throw new IllegalTransactionStateException(
            "Existing transaction found for transaction marked with
propagation
            'never'");
        }
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_NOT_
        SUPPORTED) {
            if (debugEnabled) {
                logger.debug("Suspending current transaction");
            }
            Object suspendedResources = suspend(transaction);
            boolean newSynchronization = (getTransactionSynchronization()
== SYNCHRONIZATION_
            ALWAYS);
        return prepareTransactionStatus(
        definition, null, false, newSynchronization, debugEnabled,
        suspendedResources);
    }
    if (definition.getPropagationBehavior() ==
```

```
TransactionDefinition.PROPAGATION_
        REQUIRES_NEW) {
            if (debugEnabled) {
                logger.debug("Suspending current transaction, creating new
transaction
                with name [" +
                definition.getName() + "]");
            }
            //新事务的建立
            SuspendedResourcesHolder suspendedResources =
suspend(transaction);
            try {
                boolean newSynchronization = (getTransactionSynchronization()
!=
               SYNCHRONIZATION_NEVER);
                DefaultTransactionStatus status = newTransactionStatus(
                definition, transaction, true, newSynchronization, debugEnabled,
               suspendedResources);
                doBegin(transaction, definition);
                prepareSynchronization(status, definition);
                return status;
            }
            catch (RuntimeException beginEx) {
                resumeAfterBeginException(transaction, suspendedResources,
beginEx);
                throw beginEx;
            }
```

```
            catch (Error beginErr) {
                resumeAfterBeginException(transaction, suspendedResources,
beginErr);
                throw beginErr;
            }
        }
        //嵌入式事务的处理
        if (definition.getPropagationBehavior() ==
TransactionDefinition.PROPAGATION_
        NESTED) {
            if (!isNestedTransactionAllowed()) {
                throw new NestedTransactionNotSupportedException(
                "Transaction manager does not allow nested transactions by
                default - " +
                "specify 'nestedTransactionAllowed' property with value 'true'");
            }
            if (debugEnabled) {
                logger.debug("Creating nested transaction with name [" +
                definition.getName() + "]");
            }
            if (useSavepointForNestedTransaction()) {
               //如果没有可以使用保存点的方式控制事务回滚，那么在嵌入
式事务的建立初始建立保存点
                DefaultTransactionStatus status =
                prepareTransactionStatus(definition, transaction, false,
                false, debugEnabled, null);
               status.createAndHoldSavepoint();
```

```
            return status;
          }else {
           //有些情况是不能使用保存点操作，比如JTA，那么建立新事
物
            boolean newSynchronization = (getTransactionSynchronization()
!=
           SYNCHRONIZATION_NEVER);
            DefaultTransactionStatus status = newTransactionStatus(
           definition,transaction, true, newSynchronization,
            debugEnabled, null);
            doBegin(transaction, definition);
            prepareSynchronization(status, definition);
            return status;
          }
      }
      if (debugEnabled) {
          logger.debug("Participating in existing transaction");
      }
      if (isValidateExistingTransaction()) {
          if (definition.getIsolationLevel() !=
TransactionDefinition.PROPAGATION_
          REQUIRES_NEW) {
            Integer currentIsolationLevel =
TransactionSynchronizationManager.
           getCurrentTransactionIsolationLevel();
            if (currentIsolationLevel == null || currentIsolationLevel !=
            definition.getIsolationLevel()) {
```

```
                    Constants isoConstants =
DefaultTransactionDefinition.constants;
                    throw new IllegalTransactionStateException("Participating
transaction
                    with definition [" +
                    definition + "] specifies isolation level which is
                    incompatible with existing transaction: " +
                    (currentIsolationLevel != null ?
                   isoConstants.toCode(currentIsolationLevel,
                    DefaultTransactionDefinition.PREFIX_ISOLATION) :
                   "(unknown)"));
                }
            }
            if (!definition.isReadOnly()) {
                if

(TransactionSynchronizationManager.isCurrentTransactionReadOnly()) {
                    throw new IllegalTransactionStateException("Participating
transaction
                    with definition [" +
                    definition + "] is not marked as read-only but existing
                    transaction is");
                }
            }
            }
            boolean newSynchronization = (getTransactionSynchronization() !=
SYNCHRONIZATION_
```

NEVER);
return prepareTransactionStatus(definition, transaction, false, newSynchronization,
debugEnabled, null);
}

对于已经存在事务的处理过程中，我们看到了很多熟悉的操作，但是，也有些不同的地方，函数中对已经存在的事务处理考虑两种情况。

（1）PROPAGATION_REQUIRES_NEW 表示当前方法必须在它自己的事务里运行，一个新的事务将被启动，而如果有一个事务正在运行的话，则在这个方法运行期间被挂起。而Spring中对于此种传播方式的处理与新事务建立最大的不同点在于使用suspend方法将原事务挂起。将信息挂起的目的当然是为了在当前事务执行完毕后在将原事务还原。

（2）PROPAGATION_NESTED 表示如果当前正有一个事务在运行中，则该方法应该运行在一个嵌套的事务中，被嵌套的事务可以独立于封装事务进行提交或者回滚，如果封装事务不存在，行为就像PROPAGATION_REQUIRES_NEW。对于嵌入式事务的处理，Spring 中主要考虑了两种方式的处理。

Spring中允许嵌入事务的时候，则首选设置保存点的方式作为异常处理的回滚。

对于其他方式，比如 JTA 无法使用保存点的方式，那么处理方式与 PROPAGATION_REQUIRES_NEW相同，而一旦出现异常，则由Spring的事务异常处理机制去完成后续操作。

对于挂起操作的主要目的是记录原有事务的状态，以便于后续操作对事务的恢复：

protected final SuspendedResourcesHolder suspend(Object transaction) throws Transaction

Exception {

```
        if (TransactionSynchronizationManager.isSynchronizationActive()) {
            List<TransactionSynchronization> suspendedSynchronizations =
doSuspend
           Synchronization();
            try {
                Object suspendedResources = null;
                if (transaction != null) {
                    suspendedResources = doSuspend(transaction);
                }
                String name = TransactionSynchronizationManager. GetCurrent
Transaction
               Name();

TransactionSynchronizationManager.setCurrentTransactionName(null);
                boolean readOnly =
TransactionSynchronizationManager.isCurrentTransaction
               ReadOnly();

TransactionSynchronizationManager.setCurrentTransactionReadOnly(false);
                Integer isolationLevel =
TransactionSynchronizationManager.getCurrent
               TransactionIsolationLevel();

TransactionSynchronizationManager.setCurrentTransactionIsolation
               Level(null);
                boolean wasActive =
TransactionSynchronizationManager.isActual
```

```
                TransactionActive();

TransactionSynchronizationManager.setActualTransactionActive(false);
                return new SuspendedResourcesHolder(
                suspendedResources, suspendedSynchronizations, name,
readOnly,
                isolationLevel, wasActive);
            }
            catch (RuntimeException ex) {
                // doSuspend failed - original transaction is still active...
                doResumeSynchronization(suspendedSynchronizations);
                throw ex;
            }
            catch (Error err) {
                doResumeSynchronization(suspendedSynchronizations);
                throw err;
            }
        }
        else if (transaction != null) {
            Object suspendedResources = doSuspend(transaction);
            return new SuspendedResourcesHolder(suspendedResources);
        }
        else {
            return null;
        }
    }
```

3．准备事务信息

当已经建立事务连接并完成了事务信息的提取后，我们需要将所有的事务信息统一记录在TransactionInfo 类型的实例中，这个实例包含了目标方法开始前的所有状态信息，一旦事务执行失败，Spring会通过TransactionInfo类型的实例中的信息来进行回滚等后续工作。

```
protected TransactionInfo
prepareTransactionInfo(PlatformTransactionManager tm,
    TransactionAttribute txAttr, String joinpointIdentification, Transaction
    Status status) {
        TransactionInfo txInfo = new TransactionInfo(tm, txAttr,
joinpointIdentification);
        if (txAttr != null) {
            // We need a transaction for this method
            if (logger.isTraceEnabled()) {
                logger.trace("Getting transaction for [" + txInfo.getJoinpoint
                Identification() + "]");
            }
            //记录事务状态
            txInfo.newTransactionStatus(status);
        }
        else {
            if (logger.isTraceEnabled())
            logger.trace("Don't need to create transaction for [" + joinpoint
            Identification +
            "]: This method isn't transactional.");
        }
        txInfo.bindToThread();
        return txInfo;
```

}

10.3.2 回滚处理

之前已经完成了目标方法运行前的事务准备工作，而这些准备工作最大的目的无非是对于程序没有按照我们期待的那样进行，也就是出现特定的错误，那么，当出现错误的时候，Spring是怎么对数据进行恢复的呢？

```
protected void completeTransactionAfterThrowing(TransactionInfo
txInfo, Throwable ex) {
    //当抛出异常时首先判断当前是否存在事务，这是基础依据
     if (txInfo != null && txInfo.hasTransaction()) {
        if (logger.isTraceEnabled()) {
            Identification() +
            logger.trace("Completing transaction for [" + txInfo.getJoinpoint
            "] after exception: " + ex);
        }
        //这里判断是否回滚默认的依据是抛出的异常是否是
RuntimeException或者是Error的类型
        if (txInfo.transactionAttribute.rollbackOn(ex)) {
            try {
              //根据TransactionStatus信息进行回滚处理

txInfo.getTransactionManager().rollback(txInfo.GetTransaction
              Status());
           }
            catch (TransactionSystemException ex2) {
               logger.error("Application exception overridden by rollback
```

```
                        exception", ex);
                      ex2.initApplicationException(ex);
                        throw ex2;
                    }
                    catch (RuntimeException ex2) {
                        logger.error("Application exception overridden by rollback
                        exception", ex);
                        throw ex2;
                    }
                    catch (Error err) {
                        logger.error("Application exception overridden by rollback
error", ex);
                        throw err;
                    }
                }else {
                   //如果不满足回滚条件即使抛出异常也同样会提交
                    try {

txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
                    }
                    catch (TransactionSystemException ex2) {
                        logger.error("Application exception overridden by commit
                        exception", ex);
                      ex2.initApplicationException(ex);
                        throw ex2;
                    }
                    catch (RuntimeException ex2) {
```

```
                logger.error("Application exception overridden by commit
                exception", ex);
                throw ex2;
            }
            catch (Error err) {
                logger.error("Application exception overridden by commit
error", ex);
                throw err;
            }
        }
    }
}
```

在对目标方法的执行过程中，一旦出现Throwable就会被引导至此方法处理，但是并不代表所有的 Throwable 都会被回滚处理，比如我们最常用的 Exception，默认是不会被处理的。默认情况下，即使出现异常，数据也会被正常提交，而这个关键的地方就是在txInfo.transactionAttribute.rollbackOn(ex)这个函数。

1．回滚条件

```
public boolean rollbackOn(Throwable ex) {
    return (ex instanceof RuntimeException || ex instanceof Error);
}
```

看到了吗?默认情况下Spring中的事务异常处理机制只对RuntimeException和Error两种情况感兴趣，当然你可以通过扩展来改变，不过，我们最常用的还是使用事务提供的属性设置，利用注解方式的使用，例如：

```
@Transactional(propagation=Propagation.REQUIRED,rollbackFor=Exc
```

2．回滚处理

当然，一旦符合回滚条件，那么Spring就会将程序引导至回滚处理函数中。

```
public final void rollback(TransactionStatus status) throws
TransactionException {
    //如果事务已经完成，那么再次回滚会抛出异常
     if (status.isCompleted()) {
        throw new IllegalTransactionStateException(
        "Transaction is already completed - do not call commit or rollback
        more than once per transaction");
     }
     DefaultTransactionStatus defStatus = (DefaultTransactionStatus)
status;
    processRollback(defStatus);
}
private void processRollback(DefaultTransactionStatus status) {
     try {
        try {
          //激活所有TransactionSynchronization中对应的方法
          triggerBeforeCompletion(status);
           if (status.hasSavepoint()) {
              if (status.isDebug()) {
                 logger.debug("Rolling back transaction to savepoint");
              }
             //如果有保存点，也就是当前事务为单独的线程则会退到
保存点
             status.rollbackToHeldSavepoint();
          }
```

```
else if (status.isNewTransaction()) {
    if (status.isDebug()) {
        logger.debug("Initiating transaction rollback");
    }
    //如果当前事务为独立的新事物，则直接回退
    doRollback(status);
}
else if (status.hasTransaction()) {
    if (status.isLocalRollbackOnly() ||
isGlobalRollbackOnParticipation
    Failure()) {
    if (status.isDebug()) {
        logger.debug("Participating transaction failed -
        marking existing transaction as rollback-only");
    }
    //如果当前事务不是独立的事务，那么只能标记状态，等
到事务链执行完毕后统
    一回滚
    doSetRollbackOnly(status);
}
else {
    if (status.isDebug()) {
        logger.debug("Participating transaction failed -
        letting transaction originator decide on rollback");
    }
}
}
```

```
            else {
                logger.debug("Should roll back transaction but cannot - no
                transaction available");
            }
        }
        catch (RuntimeException ex) {
            triggerAfterCompletion(status,
TransactionSynchronization.STATUS_UNKNOWN);
            throw ex;
        }
        catch (Error err) {
            triggerAfterCompletion(status,
TransactionSynchronization.STATUS_UNKNOWN);
            throw err;
        }
        //激活所有TransactionSynchronization中对应的方法
        triggerAfterCompletion(status,
TransactionSynchronization.STATUS_ROLLED_BACK);
        }
        finally {
        //清空记录的资源并将挂起的资源恢复
        cleanupAfterCompletion(status);
        }
    }
```

同样，对于在Spring中的复杂的逻辑处理过程，在入口函数一般都会给出个整体的处理脉络，而把实现细节委托给其他函数去执行。我们尝试总结下Spring中对于回滚处理的大致脉络如下。

（1）首先是自定义触发器的调用，包括在回滚前、完成回滚后的调用，当然完成回滚包括正常回滚与回滚过程中出现异常，自定义的触发器会根据这些信息作进一步处理，而对于触发器的注册，常见是在回调过程中通过TransactionSynchronizationManager类中的静态方法直接注册：

```
public static void registerSynchronization(TransactionSynchronization
synchronization)
```

（2）除了触发监听函数外，就是真正的回滚逻辑处理了。

当之前已经保存的事务信息中有保存点信息的时候，使用保存点信息进行回滚。常用于嵌入式事务，对于嵌入式的事务的处理，内嵌的事务异常并不会引起外部事务的回滚。

根据保存点回滚的实现方式其实是根据底层的数据库连接进行的。

```
public void rollbackToHeldSavepoint() throws TransactionException {
    if (!hasSavepoint()) {
        throw new TransactionUsageException("No savepoint associated
with current
        transaction");
    }
    getSavepointManager().rollbackToSavepoint(getSavepoint());
    setSavepoint(null);
}
```

这里使用的是 JDBC 的方式进行数据库连接，那么getSavepointManager()函数返回的是JdbcTransactionObjectSupport，也就是说上面函数会调用 JdbcTransactionObjectSupport 中的rollbackToSavepoint方法。

```
public void rollbackToSavepoint(Object savepoint) throws
TransactionException {
```

```
            try {

getConnectionHolderForSavepoint().getConnection().rollback((Savepoint)
               savepoint);
            }
            catch (Throwable ex) {
                throw new TransactionSystemException("Could not roll back to
JDBC savepoint", ex);
            }
        }
```

当之前已经保存的事务信息中的事务为新事物，那么直接回滚。常用于单独事务的处理。对于没有保存点的回滚，Spring同样是使用底层数据库连接提供的API来操作的。由于我们使用的是DataSourceTransactionManager，那么doRollback函数会使用此类中的实现：

```
        protected void doRollback(DefaultTransactionStatus status) {
            DataSourceTransactionObject txObject =
(DataSourceTransactionObject)
           status.getTransaction();
            Connection con = txObject.getConnectionHolder().getConnection();
            if (status.isDebug()) {
                logger.debug("Rolling back JDBC transaction on Connection [" +
con + "]");
            }
            try {
               con.rollback();
            }
```

```
    catch (SQLException ex) {
        throw new TransactionSystemException("Could not roll back
JDBC transaction", ex);
    }
}
```

当前事务信息中表明是存在事务的，又不属于以上两种情况，多数用于JTA，只做回滚标识，等到提交的时候统一不提交。

3．回滚后的信息清除

对于回滚逻辑执行结束后，无论回滚是否成功，都必须要做的事情就是事务结束后的收尾工作。

```
private void cleanupAfterCompletion(DefaultTransactionStatus status) {
    //设置完成状态
    status.setCompleted();
    if (status.isNewSynchronization()) {
        TransactionSynchronizationManager.clear();
    }
    if (status.isNewTransaction()) {
        doCleanupAfterCompletion(status.getTransaction());
    }
    if (status.getSuspendedResources() != null) {
        if (status.isDebug()) {
            logger.debug("Resuming suspended transaction after completion
of inner
            transaction");
        }
        //结束之前事务的挂起状态
        resume(status.getTransaction(), (SuspendedResourcesHolder)
```

status.getSuspended
            Resources());
        }
    }

从函数中得知，事务处理的收尾处理工作包括如下内容。

（1）设置状态是对事务信息作完成标识以避免重复调用。

（2）如果当前事务是新的同步状态，需要将绑定到当前线程的事务信息清除。

（3）如果是新事物需要做些清除资源的工作。

```
protected void doCleanupAfterCompletion(Object transaction) {
    DataSourceTransactionObject txObject =
(DataSourceTransactionObject) transaction;
    if (txObject.isNewConnectionHolder()) {
      //将数据库连接从当前线程中解除绑定

TransactionSynchronizationManager.unbindResource(this.dataSource);
    }
    //释放链接
    Connection con = txObject.getConnectionHolder().getConnection();
    try {
       if (txObject.isMustRestoreAutoCommit()) {
         //恢复数据库连接的自动提交属性
         con.setAutoCommit(true);
       }
      //重置数据库连接
       DataSourceUtils.resetConnectionAfterTransaction(con,
txObject.getPrevious
```

```
            IsolationLevel());
        }
        catch (Throwable ex) {
            logger.debug("Could not reset JDBC Connection after
transaction", ex);
        }
        if (txObject.isNewConnectionHolder()) {
            if (logger.isDebugEnabled()) {
                logger.debug("Releasing JDBC Connection [" + con + "] after
                transaction");
            }
            //如果当前事务时独立的新创建的事务则在事务完成时释放数
据库连接
            DataSourceUtils.releaseConnection(con, this.dataSource);
        }
        txObject.getConnectionHolder().clear();
    }
```

（4）如果在事务执行前有事务挂起，那么当前事务执行结束后需要将挂起事务恢复。

```
    protected final void resume(Object transaction,
SuspendedResourcesHolder resourcesHolder)
    throws TransactionException {
        if (resourcesHolder != null) {
            Object suspendedResources =
resourcesHolder.suspendedResources;
            if (suspendedResources != null) {
                doResume(transaction, suspendedResources);
```

```
            }
             List<TransactionSynchronization> suspendedSynchronizations =
resourcesHolder.
            suspendedSynchronizations;
             if (suspendedSynchronizations != null) {

TransactionSynchronizationManager.setActualTransactionActive (resources
               Holder.wasActive);

TransactionSynchronizationManager.setCurrentTransactionIsolationLevel
               (resourcesHolder.isolationLevel);

TransactionSynchronizationManager.setCurrentTransactionReadOnly
               (resourcesHolder.readOnly);

TransactionSynchronizationManager.setCurrentTransactionName
               (resourcesHolder.name);
               doResumeSynchronization(suspendedSynchronizations);
            }
         }
      }
```

10.3.3 事务提交

之前我们分析了Spring的事务异常处理机制，那么事务的执行并没有出现任何的异常，也就意味着事务可以走正常事务提交的流程了。

```
protected void commitTransactionAfterReturning(TransactionInfo
txInfo) {
```

```
        if (txInfo != null && txInfo.hasTransaction()) {
            if (logger.isTraceEnabled()) {
                logger.trace("Completing transaction for [" + txInfo.getJoinpoint
                Identification() + "]");
            }

txInfo.getTransactionManager().commit(txInfo.getTransactionStatus());
        }
    }
```

在真正的数据提交之前，还需要做个判断。不知道大家还有没有印象，在我们分析事务异常处理规则的时候，当某个事务既没有保存点又不是新事物，Spring对它的处理方式只是设置一个回滚标识。这个回滚标识在这里就会派上用场了，主要的应用场景如下。

某个事务是另一个事务的嵌入事务，但是，这些事务又不在Spring的管理范围内，或者无法设置保存点，那么 Spring 会通过设置回滚标识的方式来禁止提交。首先当某个嵌入事务发生回滚的时候会设置回滚标识，而等到外部事务提交时，一旦判断出当前事务流被设置了回滚标识，则由外部事务来统一进行整体事务的回滚。

所以，当事务没有被异常捕获的时候也并不意味着一定会执行提交的过程。

```
    public final void commit(TransactionStatus status) throws
TransactionException {
        if (status.isCompleted()) {
            throw new IllegalTransactionStateException(
            "Transaction is already completed - do not call commit or rollback
            more than once per transaction");
        }
```

```
    DefaultTransactionStatus defStatus = (DefaultTransactionStatus)
status;
    //如果在事务链中已经被标记回滚，那么不会尝试提交事务，直
接回滚
    if (defStatus.isLocalRollbackOnly()) {
        if (defStatus.isDebug()) {
            logger.debug("Transactional code has requested rollback");
        }
        processRollback(defStatus);
        return;
    }
    if (!shouldCommitOnGlobalRollbackOnly() &&
defStatus.isGlobalRollbackOnly()) {
        if (defStatus.isDebug()) {
            logger.debug("Global transaction is marked as rollback-only but
            transactional code requested commit");
        }
        processRollback(defStatus);
        if (status.isNewTransaction() ||
isFailEarlyOnGlobalRollbackOnly()) {
            throw new UnexpectedRollbackException(
            "Transaction rolled back because it has been marked as
            rollback-only");
        }
        return;
    }
    //处理事务提交
```

```
      processCommit(defStatus);
    }
```

而当事务执行一切都正常的时候，便可以真正地进入提交流程了。

```
    private void processCommit(DefaultTransactionStatus status) throws
TransactionException {
        try {
            boolean beforeCompletionInvoked = false;
            try {
              //预留
              prepareForCommit(status);
              //添加的TransactionSynchronization中的对应方法的调用
              triggerBeforeCommit(status);
              //添加的TransactionSynchronization中的对应方法的调用
              triggerBeforeCompletion(status);
               beforeCompletionInvoked = true;
               boolean globalRollbackOnly = false;
               if (status.isNewTransaction() ||
isFailEarlyOnGlobalRollbackOnly()) {
                   globalRollbackOnly = status.isGlobalRollbackOnly();
              }
               if (status.hasSavepoint()) {
                   if (status.isDebug()) {
                       logger.debug("Releasing transaction savepoint");
                  }
                  //如果存在保存点则清除保存点信息
                  status.releaseHeldSavepoint();
              }
```

```
            else if (status.isNewTransaction()) {
                if (status.isDebug()) {
                    logger.debug("Initiating transaction commit");
                }
                //如果是独立的事务则直接提交
                doCommit(status);
            }
            if (globalRollbackOnly) {
                throw new UnexpectedRollbackException(
                "Transaction silently rolled back because it has been
                marked as rollback-only");
            }
        }
        catch (UnexpectedRollbackException ex) {
            triggerAfterCompletion(status,
TransactionSynchronization.STATUS_
            ROLLED_BACK);
            throw ex;
        }
        catch (TransactionException ex) {
            if (isRollbackOnCommitFailure()) {
                doRollbackOnCommitException(status, ex);
            }
            else {
                triggerAfterCompletion(status,
TransactionSynchronization.STATUS_
                UNKNOWN);
```

```
        }
        throw ex;
    }
    catch (RuntimeException ex) {
        if (!beforeCompletionInvoked) {
            triggerBeforeCompletion(status);
        }
        doRollbackOnCommitException(status, ex);
        throw ex;
    }
    catch (Error err) {
        if (!beforeCompletionInvoked) {
            //添加的TransactionSynchronization中的对应方法的调用
            triggerBeforeCompletion(status);
        }
        //提交过程中出现异常则回滚
        doRollbackOnCommitException(status, err);
        throw err;
    }
    try {
        //添加的TransactionSynchronization中的对应方法的调用
        triggerAfterCommit(status);
    }
    finally {
        triggerAfterCompletion(status,
TransactionSynchronization.STATUS_
        COMMITTED);
```

```
            }
        }
        finally {
            cleanupAfterCompletion(status);
        }
    }
```

在提交过程中也并不是直接提交的，而是考虑了诸多的方面，符合提交的条件如下。

当事务状态中有保存点信息的话便不会去提交事务。

当事务非新事务的时候也不会去执行提交事务操作。

此条件主要考虑内嵌事务的情况，对于内嵌事务，在Spring中正常的处理方式是将内嵌事务开始之前设置保存点，一旦内嵌事务出现异常便根据保存点信息进行回滚，但是如果没有出现异常，内嵌事务并不会单独提交，而是根据事务流由最外层事务负责提交，所以如果当前存在保存点信息便不是最外层事务，不做保存操作，对于是否是新事务的判断也是基于此考虑。

如果程序流通过了事务的层层把关，最后顺利地进入了提交流程，那么同样，Spring会将事务提交的操作引导至底层数据库连接的API，进行事务提交。

```
    protected void doCommit(DefaultTransactionStatus status) {
        DataSourceTransactionObject txObject =
(DataSourceTransactionObject) status.
        getTransaction();
        Connection con = txObject.getConnectionHolder().getConnection();
        if (status.isDebug()) {
            logger.debug("Committing JDBC transaction on Connection [" +
con + "]");
```

```
        }
         try {
            con.commit();
        }
         catch (SQLException ex) {
            throw new TransactionSystemException("Could not commit JDBC transaction", ex);
        }
    }
```

第11章　SpringMVC

Spring框架提供了构建Web应用程序的全功能MVC模块。通过策略接口，Spring框架是高度可配置的，而且支持多种视图技术，例如JavaServer Pages（JSP）技术、Velocity、Tiles、iText和POI。Spring MVC框架并不知道使用的视图，所以不会强迫您只使用 JSP技术。Spring MVC 分离了控制器、模型对象、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制。

Spring的MVC是基于Servlet功能实现的，通过实现Servlet接口的DispatcherServlet来封装其核心功能实现，通过将请求分派给处理程序，同时带有可配置的处理程序映射、视图解析、本地语言、主题解析以及上载文件支持。默认的处理程序是非常简单的 Controller 接口，只有一个方法ModelAndView handleRequest(request, response)。Spring提供了一个控制器层次结构，可以派生子类。如果应用程序需要处理用户输入表单，那么可以继承AbstractFormController。如果需要把多页输入处理到一个表单，那么可以继承AbstractWizardFormController。

SpringMVC或者其他比较成熟的MVC框架而言，解决的问题无外

乎以下几点。

（1）将Web页面的请求传给服务器。

（2）根据不同的请求处理不同的逻辑单元。

（3）返回处理结果数据并跳转至响应的页面。

我们首先通过一个简单示例来快速回顾SpringMVC的使用。

11.1 SpringMVC快速体验

（1）配置web.xml。

一个Web中可以没有web.xml文件，也就是说，web.xml文件并不是Web工程必须的。web.xml文件用来初始化配置信息：比如Welcome页面、servlet、servlet-mapping、filter、listener、启动加载级别等。但是，SpringMVC的实现原理是通过Servlet拦截所有URL来达到控制的目的，所以web.xml的配置是必须的。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.5"
xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.
    com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">
      <display-name>Springmvc</display-name>
         <!--使用ContextLoaderListener配置时，需要告诉它Spring配置
文件的位置 -->
      <context-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>classpath:applicationContext.xml</param-value>
```

```
</context-param>
 <!-- SpringMVC的前端控制器 -->
<!-- 当DispatcherServlet载入后，它将从一个XML文件中载入
Spring的应用上下文，该XML文件的名
字取决于<servlet-name> -->
<!-- 这里DispatcherServlet将试图从一个叫做Springmvc-servlet.xml
的文件中载入应用上下文，其
默认位于WEB-INF目录下 -->
<servlet>
  <servlet-name>Springmvc</servlet-name>
  <servlet-
class>org.Springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Springmvc</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
 <!-- 配置上下文载入器 -->
 <!-- 上下文载入器载入除DispatcherServlet载入的配置文件之外
的其它上下文配置文件 -->
 <!-- 最常用的上下文载入器是一个Servlet监听器，其名称为
ContextLoaderListener -->
<listener>
  <listener-
class>org.Springframework.web.context.ContextLoaderListener
  </listener-class>
```

```
    </listener>
</web-app>
```

Spring的MVC之所以必须要配置web.xml，其实最关键的是要配置两个地方。

contextConfigLocation：Spring的核心就是配置文件，可以说配置文件是Spring中必不可少的东西，而这个参数就是使Web与Spring的配置文件相结合的一个关键配置。

DispatcherServlet：包含了SpringMVC的请求逻辑，Spring使用此类拦截Web请求并进行相应的逻辑处理。

（2）创建Spring配置文件applicationContext.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.Springframework.org/schema/tx"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
    http://www.Springframework.org/schema/beans/Spring-beans-2.5.xsd
    http://www.Springframework.org/schema/tx
    http://www.Springframework.org/schema/tx/Spring-tx-2.5.xsd">
     <bean id="viewResolver"
class="org.Springframework.web.servlet.view. Internal Resource
    ViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>
</beans>
```

InternalResourceViewResolver是一个辅助Bean，会在ModelAndView

返回的视图名前加上prefix 指定的前缀，再在最后加上 suffix 指定的后缀，例如：由于 XXController 返回的ModelAndView 中的视图名是testview，故该视图解析器将在/WEB-INF/jsp/testview.jsp 处查找视图。

（3）创建model。

模型对于 SpringMVC 来说并不是必不可少，如果处理程序非常简单，完全可以忽略。模型创建主要的目的就是承载数据，使数据传输更加方便。

```
public class User {
    private String username;
private Integer age;
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
}
```

（4）创建controller。

控制器用于处理Web请求，每个控制器都对应着一个逻辑处理。

```
public class UserController extends AbstractController {
   @Override
```

```
protected ModelAndView handleRequestInternal(HttpServletRequest
arg0, HttpServletResponse
arg1) throws Exception {
    List<User> userList = new ArrayList<User>();
    User userA = new User();
    User userB = new User();
   userA.setUsername("张三");
   userA.setAge(27);
   userB.setUsername("李四");
   userB.setAge(37);
   userList.add(userA);
   userList.add(userB);
    return new ModelAndView("userlist", "users", userList);
   }
}
```

在请求的最后返回了ModelAndView类型的实例。ModelAndView类在SpringMVC中占有很重要的地位， 控制器执行方法都必须返回一个ModelAndView，ModelAndView对象保存了视图以及视图显示的模型数据，例如其中的参数如下。

第一个参数userlist：视图组件的逻辑名称。这里视图的逻辑名称就是userlist，视图解析器会使用该名称查找实际的View对象。

第二个参数users：传递给视图的，模型对象的名称。

第三个参数userList：传递给视图的，模型对象的值。

（5）创建视图文件userlist.jsp。

```
<%@ page language="java" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<h2>This is SpringMVC demo page</h2>
```

```
<c:forEach items="${users}" var="user">
    <c:out value="${user.username}"/><br/>
    <c:out value="${user.age}"/><br/>
</c:forEach>
```

视图文件用于展现请求处理结果，通过对JSTL的支持，可以很方便地展现在控制器中放入ModelAndView中的处理结果数据。

（6）创建Servlet配置文件Spring-servlet.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tx="http://www. Spring
    framework. org/schema/tx"

xsi:schemaLocation="http://www.Springframework.org/schema/beans
http://www.Springframework.org/schema/beans/Spring-beans-2.5.xsd
http://www.Springframework.org/schema/tx
http://www.Springframework.org/schema/tx/Spring-tx-2.5.xsd">
    <bean id="simpleUrlMapping"

class="org.Springframework.web.servlet.handler.SimpleUrlHandlerMapping"
        <property name="mappings">
          <props>
              <prop key="/userlist.htm">userController</prop>
          </props>
        </property>
      </bean>
    <!-- 这里的 id="userController"对应的是<bean
```

id="simpleUrlMapping">中的<prop>里面的

value -->

<bean id="userController" class="test.controller.UserController" />

</beans>

因为SpringMVC是基于Servlet的实现，所以在Web启动的时候，服务器会首先尝试加载对应于Servlet的配置文件，而为了让项目更加模块化，通常我们将Web部分的配置都存放于此配置文件中。

至此，已经完成了 SpringMVC 的搭建，启动服务器，输入网址 http://localhost:8080/Springmvc/userlist.htm。

看到了服务器返回界面，如图11-1所示。



图11-1 Spring MVC快速体验

11.2 ContextLoaderListener

对于SpringMVC功能实现的分析，我们首先从web.xml开始，在web.xml文件中我们首先配置的就是ContextLoaderListener，那么它所提供的功能有哪些又是如何实现的呢？

当使用编程方式的时候我们可以直接将Spring配置信息作为参数传入Spring容器中，如

ApplicationContext ac=new

ClassPathXmlApplicationContext(“applicationContext.xml”);

但是在 Web 下，我们需要更多的是与 Web 环境相互结合，通常的办法是将路径以context-param的方式注册并使用ContextLoaderListener进行监听读取。

ContextLoaderListener的作用就是启动Web容器时，自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，在web.xml配置这个监听器，启动容器时，就会默认执行它实现的方法，使用ServletContextListener接口，开发者能够在为客户端请求提供服务之前向ServletContext中添加任意的对象。这个对象在ServletContext启动的时候被初始化，然后在ServletContext整个运行期间都是可见的。

每一个Web应用都有一个ServletContext与之相关联。ServletContext对象在应用启动时被创建，在应用关闭的时候被销毁。ServletContext在全局范围内有效，类似于应用中的一个全局变量。

在 ServletContextListener 中的核心逻辑便是初始化WebApplicationContext 实例并存放至ServletContext中。

11.2.1 ServletContextListener的使用

正式分析代码前我们同样还是首先了解ServletContextListener的使用。

（1）创建自定义ServletContextListener。

首先我们创建 ServletContextListener，目标是在系统启动时添加自定义的属性，以便于在全局范围内可以随时调用。系统启动的时候会调用 ServletContextListener 实现类的contextInitialized方法，所以需要在这个方法中实现我们的初始化逻辑。

public class MyDataContextListener implements ServletContextListener
{

```
    private ServletContext context = null;
    public MyDataContextListener () {
    }
    //该方法在ServletContext启动之后被调用，并准备好处理客户端请求
    public void contextInitialized(ServletContextEvent event) {
this.context = event.getServletContext();
//通过你可以实现自己的逻辑并将结果记录在属性中
    context = setAttribute("myData","this is myData");
    }
    //这个方法在ServletContext将要关闭的时候调用
    public void contextDestroyed(ServletContextEvent event){
        this.context = null;
    }
}
```

（2）注册监听器。

在web.xml文件中需要注册自定义的监听器。

```
<listener>
com.test.MyDataContextListener
</listener>
```

（3）测试。

一旦Web应用启动的时候，我们就能在任意的Servlet或者JSP中通过下面的方式获取我们初始化的参数，如下：

```
String myData = (String) getServletContext().getAttribute("myData");
```

11.2.2 Spring中的ContextLoaderListener

分析了ServletContextListener的使用方式后再来分析Spring中的

ContextLoaderListener的实现就容易理解的多，虽然ContextLoaderListener实现的逻辑要复杂的多，但是大致的套路还是万变不离其宗。

ServletContext启动之后会调用ServletContextListener的contextInitialized方法，那么，我们就从这个函数开始进行分析。

```
public void contextInitialized(ServletContextEvent event) {
    this.contextLoader = createContextLoader();
    if (this.contextLoader == null) {
        this.contextLoader = this;
    }
//初始化WebApplicationContext

this.contextLoader.initWebApplicationContext(event.getServletContext());
}
```

这里涉及了一个常用类WebApplicationContext：在Web应用中，我们会用到WebApplication Context，WebApplicationContext继承自ApplicationContext，在ApplicationContext的基础上又追加了一些特定于Web 的操作及属性，非常类似于我们通过编程方式使用 Spring 时使用的ClassPathXmlApplicationContext类提供的功能。继续跟踪代码：

```
public WebApplicationContext
initWebApplicationContext(ServletContext servletContext) {
    if
(servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICA
   CONTEXT_ATTRIBUTE) != null) {
//web.xml中存在多次ContextLoader定义
    throw new IllegalStateException(
    "Cannot initialize context because there is already a root
```

```
            application context present - " +
           web.xml!");
            "check whether you have multiple ContextLoader* definitions in
your
           }
            Log logger = LogFactory.getLog(ContextLoader.class);
            servletContext.log("Initializing Spring root
WebApplicationContext");
            if (logger.isInfoEnabled()) {
                logger.info("Root WebApplicationContext: initialization started");
           }
            long startTime = System.currentTimeMillis();
            try {
                // Store context in local instance variable, to guarantee that
                // it is available on ServletContext shutdown.
                if (this.context == null) {
               //初始化context
                    this.context = createWebApplicationContext(servletContext);
               }
                if (this.context instanceof ConfigurableWebApplicationContext) {

configureAndRefreshWebApplicationContext((ConfigurableWebApplication
                    Context)this.context, servletContext);
               }
           //记录在servletContext中

servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICA
```

```
                CONTEXT_ATTRIBUTE, this.context);
                 ClassLoader ccl =
Thread.currentThread().getContextClassLoader();
                 if (ccl == ContextLoader.class.getClassLoader()) {
                     currentContext = this.context;
                 }
                 else if (ccl != null) {
                     currentContextPerThread.put(ccl, this.context);
                 }
                 if (logger.isDebugEnabled()) {
                     logger.debug("Published root WebApplicationContext as
ServletContext
                     attribute with name [" +

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRI
                     + "]");
                 }
                 if (logger.isInfoEnabled()) {
                     long elapsedTime = System.currentTimeMillis() - startTime;
                     logger.info("Root WebApplicationContext: initialization
completed in
                     " + elapsedTime + " ms");
                 }
                 return this.context;
             }
             catch (RuntimeException ex) {
                 logger.error("Context initialization failed", ex);
```

```
servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICA
            CONTEXT_ATTRIBUTE, ex);
             throw ex;
          }
          catch (Error err) {
             logger.error("Context initialization failed", err);

servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICA
            CONTEXT_ATTRIBUTE, err);
             throw err;
          }
       }
```

initWebApplicationContext 函数主要是体现了创建 WebApplicationContext 实例的一个功能架构，从函数中我们看到了初始化的大致步骤。

（1）WebApplicationContext存在性的验证。

在配置中只允许声明一次ServletContextListener，多次声明会扰乱Spring的执行逻辑，所以这里首先做的就是对此验证，在 Spring 中如果创建 WebApplicationContext 实例会记录在ServletContext 中以方便全局调用，而使用的 key 就是 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRI 所以验证的方式就是查看ServletContext实例中是否有对应key的属性。

（2）创建WebApplicationContext实例。

如果通过验证，则 Spring 将创建 WebApplicationContext 实例的工作委托给了 create WebApplicationContext函数。

protected WebApplicationContext

```
createWebApplicationContext(ServletContext sc) {
            Class<?> contextClass = determineContextClass(sc);
            if
(!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)
{
                throw new ApplicationContextException("Custom context class ["
+
                contextClass.getName() +
                "] is not of type [" +
ConfigurableWebApplicationContext.class.getName()
                + "]");
           }
            ConfigurableWebApplicationContext wac =
            (ConfigurableWebApplicationContext) BeanUtils.instantiateClass
           (contextClass);
            return wac;
        }
        protected Class<?> determineContextClass(ServletContext
servletContext) {
        //CONTEXT_CLASS_PARAM = "contextClass";
            String contextClassName =
servletContext.getInitParameter(CONTEXT_CLASS_PARAM);
            if (contextClassName != null) {
                try {
                    return ClassUtils.forName(contextClassName,
ClassUtils.getDefaultClass
                   Loader());
```

```
            }
             catch (ClassNotFoundException ex) {
                 throw new ApplicationContextException(
                 "Failed to load custom context class [" + contextClassName +
                 "]", ex);
            }
        }
         else {
            contextClassName =
defaultStrategies.getProperty(WebApplicationContext.
            class.getName());
             try {
                 return ClassUtils.forName(contextClassName,
ContextLoader.class.
                getClassLoader());
            }
             catch (ClassNotFoundException ex) {
                 throw new ApplicationContextException(
                 "Failed to load default context class [" + contextClassName
                 + "]", ex);
            }
        }
    }
```

其中，在ContextLoader类中有这样的静态代码块：

```
static {
     // Load default strategy implementations from properties file.
     // This is currently strictly internal and not meant to be customized
```

```
            // by application developers.
            try {
                //DEFAULT_STRATEGIES_PATH = "ContextLoader.properties"
            ClassPathResource resource = new
ClassPathResource(DEFAULT_STRATEGIES_PATH,
           ContextLoader.class);
                defaultStrategies =
PropertiesLoaderUtils.loadProperties(resource);
           }
            catch (IOException ex) {
                throw new IllegalStateException("Could not load
'ContextLoader.properties':
                " + ex.getMessage());
           }
       }
```

根据以上静态代码块的内容，我们推断在当前类 ContextLoader 同样目录下必定会存在属性文件ContextLoader.properties，查看后果然存在，内容如下：

```
org.Springframework.web.context.WebApplicationContext=org.Springfr
support.XmlWebApplicationContext
```

综合以上代码分析，在初始化的过程中，程序首先会读取ContextLoader 类的同目录下的属性文件ContextLoader.properties，并根据其中的配置提取将要实现WebApplicationContext接口的实现类，并根据这个实现类通过反射的方式进行实例的创建。

（3）将实例记录在servletContext中。

（4）映射当前的类加载器与创建的实例到全局变量currentContextPerThread中。

11.3 DispatcherServlet

在Spring中，ContextLoaderListener只是辅助功能，用于创建WebApplicationContext类型实例，而真正的逻辑实现其实是在DispatcherServlet中进行的，DispatcherServlet是实现servlet接口的实现类。

servlet 是一个 Java 编写的程序，此程序是基于 HTTP 协议的，在服务器端运行的（如Tomcat），是按照servlet规范编写的一个Java类。主要是处理客户端的请求并将其结果发送到客户端。servlet的生命周期是由servlet的容器来控制的，它可以分为3个阶段：初始化、运行和销毁。

（1）初始化阶段。

servlet容器加载servlet类，把servlet类的.class文件中的数据读到内存中。

servlet容器创建一个ServletConfig对象。ServletConfig对象包含了servlet的初始化配置信息。

servlet容器创建一个servlet对象。

servlet容器调用servlet对象的init方法进行初始化。

（2）运行阶段。

当 servlet 容器接收到一个请求时，servlet 容器会针对这个请求创建servletRequest 和servletResponse对象，然后调用service方法。并把这两个参数传递给service方法。service方法通过 servletRequest 对象获得请求的信息。并处理该请求。再通过 servletResponse 对象生成这个请求的响应结果。然后销毁 servletRequest 和 servletResponse 对象。我们不管这个请求是post提交的还是get提交的，最终这个请求都会由service方法来处理。

（3）销毁阶段。

当Web应用被终止时，servlet容器会先调用servlet对象的destrory方法，然后再销毁servlet对象，同时也会销毁与servlet对象相关联的servletConfig对象。我们可以在destroy方法的实现中，释放servlet所占用的资源，如关闭数据库连接，关闭文件输入输出流等。

servlet的框架是由两个Java包组成：javax.servlet和javax.servlet.http。在javax.servlet包中定义了所有的servlet类都必须实现或扩展的通用接口和类，在javax.servlet.http包中定义了采用HTTP通信协议的HttpServlet类。

servlet 被设计成请求驱动，servlet 的请求可能包含多个数据项，当Web 容器接收到某个servlet请求时，servlet把请求封装成一个HttpServletRequest对象，然后把对象传给servlet的对应的服务方法。

HTTP的请求方式包括delete、get、options、post、put和trace，在HttpServlet类中分别提供了相应的服务方法，它们是doDelete()、doGet()、doOptions()、doPost()、doPut()和doTrace()。

11.3.1 servlet的使用

我们同样还是以最简单的servlet来快速体验其用法。

（1）建立servlet。

```
public class MyServlet extends HttpServlet{
    public void init(){
        System.out.println("this is init method");
    }
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        handleLogic(request,response);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
```

```
response){
        handleLogic(request,response);
      }
      private void handleLogic(HttpServletRequest request,
HttpServletResponse response){
         System.out.println("handle myLogic");
         ServletContext sc = getServletContext();
      RequestDispatcher rd = null;
      rd = sc.getRequestDispatcher("/index.jsp"); //定向的页面
      try {
         rd.forward(request, response);
      } catch (ServletException | IOException e) {
        e.printStackTrace();
      }
    }
```

麻雀虽小，五脏俱全。实例中包含了对init方法和get/post方法的处理，init方法保证在在servlet加载的时候能做一些逻辑操作，而HttpServlet类则会帮助我们根据方法类型的不同而将逻辑引入不同的函数。在子类中我们只需要重写对应的函数逻辑便可，如以上代码重写了doGet和doPost方法并将逻辑处理部分引导至handleLogic函数中，最后，又将页面跳转至index.jsp。

（2）添加配置。

为了使servlet能够正常使用，需要在web.xml文件中添加以下配置：

```
<servlet>
   <servlet-name>myservlet</servlet-name>
   <servlet-class>test.servlet.MyServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
```

</servlet>

<servlet-mapping>

<servlet-name>myservlet</servlet-name>

<url-pattern>*.htm</url-pattern>

</servlet-mapping>

配置后便可以根据对应的配置访问响应的路径了。

11.3.2 DispatcherServlet的初始化

通过上面的实例我们了解到，在servlet初始化阶段会调用其init方法，所以我们首先要查看在DispatcherServlet中是否重写了init方法。我们在其父类HttpServletBean中找到了该方法。

```
public final void init() throws ServletException {
    if (logger.isDebugEnabled()) {
        logger.debug("Initializing servlet '" + getServletName() + "'");
    }
    try {
        //解析init-param并封装只pvs中
        PropertyValues pvs = new
ServletConfigPropertyValues(getServletConfig(),
        this.requiredProperties);
    //将当前的这个Servlet类转化为一个BeanWrapper,从而能够以
Spring的方式来对init-param
    的值进行注
        BeanWrapper bw =
PropertyAccessorFactory.forBeanPropertyAccess(this);
        ResourceLoader resourceLoader = new
ServletContextResourceLoader
```

```
            (getServletContext());
            //注册自定义属性编辑器，一旦遇到Resource类型的属性将会
使用ResourceEditor进行解析
           bw.registerCustomEditor(Resource.class, new
ResourceEditor(resourceLoader,
          this.environment));
            //空实现，留给子类覆盖
            initBeanWrapper(bw);
          //属性注入
             bw.setPropertyValues(pvs, true);
           }
           catch (BeansException ex) {
             logger.error("Failed to set bean properties on servlet '" +
getServletName()
             + "'", ex);
             throw ex;
           }
       //留给子类扩展
          initServletBean();
           if (logger.isDebugEnabled()) {
             logger.debug("Servlet '" + getServletName() + "' configured
successfully");
           }
       }
```

DipatcherServlet的初始化过程主要是通过将当前的servlet类型实例转换为BeanWrapper类型实例，以便使用Spring中提供的注入功能进行对应属性的注入。这些属性如contextAttribute、contextClass、

nameSpace、contextConfigLocation等，都可以在web.xml文件中以初始化参数的方式配置在servlet的声明中。DispatcherServlet继承自FrameworkServlet，FrameworkServlet类上包含对应的同名属性，Spring会保证这些参数被注入到对应的值中。属性注入主要包含以下几个步骤。

1．封装及验证初始化参数

ServletConfigPropertyValues除了封装属性外还有对属性验证的功能。

```
public ServletConfigPropertyValues(ServletConfig config, Set<String>
requiredProperties)
throws ServletException {
    Set<String> missingProps = (requiredProperties != null &&
!requiredProperties.
    isEmpty()) ?
    new HashSet<String>(requiredProperties) : null;
    Enumeration en = config.getInitParameterNames();
    while (en.hasMoreElements()) {
        String property = (String) en.nextElement();
        Object value = config.getInitParameter(property);
        addPropertyValue(new PropertyValue(property, value));
        if (missingProps != null) {
            missingProps.remove(property);
        }
    }
    // Fail if we are still missing properties.
    if (missingProps != null && missingProps.size() > 0) {
        throw new ServletException(
```

```
            "Initialization from ServletConfig for servlet '" +
config.getServlet
            Name() +
            "' failed; the following required properties were missing: " +
            StringUtils.collectionToDelimitedString(missingProps, ", "));
    }
}
```

从代码中得知，封装属性主要是对初始化的参数进行封装，也就是servlet 中配置的<init-param>中配置的封装。当然，用户可以通过对requiredProperties参数的初始化来强制验证某些属性的必要性，这样，在属性封装的过程中，一旦检测到requiredProperties中的属性没有指定初始值，就会抛出异常。

2．将当前**servlet**实例转化成**BeanWrapper**实例

PropertyAccessorFactory.forBeanPropertyAccess是Spring中提供的工具方法，主要用于将指定实例转化为Spring中可以处理的BeanWrapper类型的实例。

3．注册相对于**Resource**的属性编辑器

属性编辑器，我们在上文中已经介绍并且分析过其原理，这里使用属性编辑器的目的是在对当前实例（ DispatcherServlet ）属性注入过程中一旦遇到 Resource 类型的属性就会使用ResourceEditor去解析。

4．属性注入

BeanWrapper为Spring中的方法，支持Spring的自动注入。其实我们最常用的属性注入无非是contextAttribute、contextClass、nameSpace、contextConfigLocation等属性。

5．**servletBean**的初始化

在ContextLoaderListener加载的时候已经创建了WebApplicationContext实例，而在这个函数中最重要的就是对这个实例

进行进一步的补充初始化。

继续查看initServletBean()。父类FrameworkServlet覆盖了HttpServletBean中的initServlet Bean函数，如下：

```
protected final void initServletBean() throws ServletException {
    getServletContext().log("Initializing Spring FrameworkServlet '" +
getServlet
    Name() + "'");
    if (this.logger.isInfoEnabled()) {
        this.logger.info("FrameworkServlet '" + getServletName() + "':
initialization
       started");
    }
    long startTime = System.currentTimeMillis();
    try {
        this.webApplicationContext = initWebApplicationContext();
    //设计为子类覆盖
       initFrameworkServlet();
    }
    catch (ServletException ex) {
        this.logger.error("Context initialization failed", ex);
        throw ex;
    }
    catch (RuntimeException ex) {
        this.logger.error("Context initialization failed", ex);
        throw ex;
    }
    if (this.logger.isInfoEnabled()) {
```

```
            long elapsedTime = System.currentTimeMillis() - startTime;
            this.logger.info("FrameworkServlet '" + getServletName() + "':
            initialization completed in " +
            elapsedTime + " ms");
        }
    }
```

上面的函数设计了计时器来统计初始化的执行时间，而且提供了一个扩展方法initFrameworkServlet()用于子类的覆盖操作，而作为关键的初始化逻辑实现委托给了initWebApplicationContext()。

11.3.3 WebApplicationContext的初始化

initWebApplicationContext 函数的主要工作就是创建或刷新 WebApplicationContext 实例并对servlet功能所使用的变量进行初始化。

```
Protected WebApplicationContext initWebApplicationContext() {
    WebApplicationContext rootContext =

WebApplicationContextUtils.getWebApplicationContext(getServletContext()
    WebApplicationContext wac = null;
    if (this.webApplicationContext != null) {
   //context实例在构造函数中被注入
        wac = this.webApplicationContext;
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac =
(ConfigurableWebApplication
            Context) wac;
            if (!cwac.isActive()) {
                if (cwac.getParent() == null) {
```

```
                cwac.setParent(rootContext);
              }
            //刷新上下文环境
              configureAndRefreshWebApplicationContext(cwac);
            }
          }
        }
        if (wac == null) {
        //根据contextAttribute属性加载WebApplicationContext
          wac = findWebApplicationContext();
        }
        if (wac == null) {
          // No context instance is defined for this servlet -> create a local
one
          wac = createWebApplicationContext(rootContext);
        }
        if (!this.refreshEventReceived) {
          // Either the context is not a ConfigurableApplicationContext with
refresh
          // support or the context injected at construction time had already
been
          // refreshed -> trigger initial onRefresh manually here.
          onRefresh(wac);
        }
        if (this.publishContext) {
          // Publish the context as a servlet context attribute.
          String attrName = getServletContextAttributeName();
```

```
            getServletContext().setAttribute(attrName, wac);
            if (this.logger.isDebugEnabled()) {
               this.logger.debug("Published WebApplicationContext of servlet
'" +
               getServletName() +
               "' as ServletContext attribute with name [" + attrName + "]");
            }
         }
      return wac;
      }
```

对于本函数中的初始化主要包含几个部分。

1．寻找或创建对应的**WebApplicationContext**实例

WebApplicationContext的寻找及创建包括以下几个步骤。

（1）通过构造函数的注入进行初始化。

当进入 initWebApplicationContext函数后通过判断this.webApplicationContext != null后，便可以确定this.webApplicationContext是否是通过构造函数来初始化的。可是有读者可能会有疑问，在initServletBean函数中明明是把创建好的实例记录在了this.webApplicationContext中：

this.webApplicationContext= initWebApplicationContext();

何以判定这个参数是通过构造函数初始化，而不是通过上一次的函数返回值初始化呢？如果存在这个问题，那么就是读者忽略一个问题了：在 Web 中包含 SpringWeb 的核心逻辑的DispatcherServlet 只可以被声明为一次，在 Spring 中已经存在验证，所以这就确保了如果this.webApplicationContext != null，则可以直接判定this.webApplicationContext已经通过构造函数初始化。

（2）通过contextAttribute进行初始化。

通过在web.xml文件中配置的servlet参数contextAttribute来查找ServletContext中对应的属性，默认为WebApplicationContext.class.getName() + ".ROOT"，也就是在ContextLoaderListener加载时会创建WebApplicationContext实例，并将实例以WebApplicationContext.class.getName() + ".ROOT"为 key 放入ServletContext 中，当然读者可以重写初始化逻辑使用自己创建的WebApplicationContext，并在servlet的配置中通过初始化参数contextAttribute指定key。

```
protected WebApplicationContext findWebApplicationContext() {
    String attrName = getContextAttribute();
    if (attrName == null) {
        return null;
    }
    WebApplicationContext wac =
      attrName);

WebApplicationContextUtils.getWebApplicationContext(getServletContext()
    if (wac == null) {
        throw new IllegalStateException("No WebApplicationContext
found: initializer
        not registered?");
    }
    return wac;
}
```

（3）重新创建WebApplicationContext实例。

如果通过以上两种方式并没有找到任何突破，那就没办法了，只能在这里重新创建新的实例了。

```
	protected WebApplicationContext
createWebApplicationContext(WebApplicationContext parent) {
		return createWebApplicationContext((ApplicationContext) parent);
	}
	protected WebApplicationContext
createWebApplicationContext(ApplicationContext parent) {
	//获取servlet的初始化参数contextClass,如果没有配置默认为
XmlWebApplicationContext.class
	Class<?> contextClass = getContextClass();
		if (this.logger.isDebugEnabled()) {
			this.logger.debug("Servlet with name '" + getServletName() +
			"' will try to create custom WebApplicationContext context of
class '" +
			contextClass.getName() + "'" + ", using parent context [" + parent
+ "]");
		}
		if
(!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass))
{
			throw new ApplicationContextException(
			"Fatal initialization error in servlet with name '" +
			getServletName() +
			"': custom WebApplicationContext class [" +
contextClass.getName() +
			"] is not of type ConfigurableWebApplicationContext");
		}
	//通过反射方式实例化contextClass
```

```
ConfigurableWebApplicationContext wac =
(ConfigurableWebApplicationContext) BeanUtils.instantiateClass
(contextClass);
//parent为在ContextLoaderListener中创建的实例
//在ContextLoaderListener加载的时候初始化的
WebApplicationContext类型实例
wac.setParent(parent);
//获取contextConfigLocation属性，配置在servlet初始化参数中
wac.setConfigLocation(getContextConfigLocation());
//初始化Spring环境包括加载配置文件等
configureAndRefreshWebApplicationContext(wac);
return wac;
}
```

2．configureAndRefreshWebApplicationContext

无论是通过构造函数注入还是单独创建，都免不了会调用configureAndRefreshWeb Application Context方法来对已经创建的WebApplicationContext实例进行配置及刷新，那么这个步骤又做了哪些工作呢？

```
protected void
configureAndRefreshWebApplicationContext(ConfigurableWebApplication
Context wac, ServletContext sc) {
    if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
        // The application context id is still set to its original default value
        // -> assign a more useful id based on available information
        String idParam = sc.getInitParameter(CONTEXT_ID_PARAM);
        if (idParam != null) {
            wac.setId(idParam);
```

```
            }
            else {
                // Generate default id...
                if (sc.getMajorVersion() == 2 && sc.getMinorVersion() < 5) {
                    // Servlet <= 2.4: resort to name specified in web.xml, if any.

wac.setId(ConfigurableWebApplicationContext.APPLICATION_
CONTEXT_
                        ID_PREFIX +
                        ObjectUtils.getDisplayString(sc.getServletContextName()));
                }
                else {

wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_
                        ID_PREFIX +
                        ObjectUtils.getDisplayString(sc.getContextPath()));
                }
            }
        }
        // Determine parent for root web application context, if any.
        ApplicationContext parent = loadParentContext(sc);
        wac.setParent(parent);
        wac.setServletContext(sc);
        String initParameter =
sc.getInitParameter(CONFIG_LOCATION_PARAM);
        if (initParameter != null) {
            wac.setConfigLocation(initParameter);
```

```
    }
     customizeContext(sc, wac);
//加载配置文件及整合parent到wac
    wac.refresh();
}
```

无论调用方式如何变化，只要是使用 AlicationContext 所提供的功能最后都免不了使用公共父类AbstractApplicationContext提供的refresh()进行配置文件加载。

3．刷新

onRefresh是FrameworkServlet类中提供的模板方法，在其子类DispatcherServlet中进行了重写，主要用于刷新Spring在Web功能实现中所必须使用的全局变量。下面我们会介绍它们的初始化过程以及使用场景，而至于具体的使用细节会在稍后的章节中再做详细介绍。

```
protected void onRefresh(ApplicationContext context) {
    initStrategies(context);
}
protected void initStrategies(ApplicationContext context) {
    //(1)初始化MultipartResolver
    initMultipartResolver(context);
    //(2)初始化LocaleResolver
    initLocaleResolver(context);
    //(3)初始化ThemeResolver
    initThemeResolver(context);
    //(4)初始化HandlerMappings
    initHandlerMappings(context);
    //(5)初始化HandlerAdapters
    initHandlerAdapters(context);
```

```
    //(6)初始化HandlerExceptionResolvers
    initHandlerExceptionResolvers(context);
    //(7)初始化RequestToViewNameTranslator
    initRequestToViewNameTranslator(context);
    //(8)初始化ViewResolvers
    initViewResolvers(context);
    //(9)初始化FlashMapManager
    initFlashMapManager(context);
}
```

（1）初始化MultipartResolver。

在Spring中，MultipartResolver主要用来处理文件上传。默认情况下，Spring是没有multipart处理的，因为一些开发者想要自己处理它们。如果想使用 Spring 的 multipart，则需要在 Web应用的上下文中添加multipart解析器。这样，每个请求就会被检查是否包含multipart。然而，如果请求中包含multipart，那么上下文中定义的MultipartResolver就会解析它，这样请求中的multipart属性就会象其他属性一样被处理。常用配置如下：

```
<bean id="multipartResolver"
class="org.Springframework.web.multipart.commons. Commons
MultipartResolver">
<!--该属性用来配置可上传文件的最大byte数 -->
<property name="maximumFileSize"><value>100000</value>
</property>
</bean>
```

当然，CommonsMultipartResolver 还提供了其他功能用于帮助用户完成上传功能，有兴趣的读者可以进一步查看。

那么MultipartResolver就是在initMultipartResolver中被加入到

DispatcherServlet中的。

```
private void initMultipartResolver(ApplicationContext context) {
    try {
        this.multipartResolver =
context.getBean(MULTIPART_RESOLVER_BEAN_NAME,
       MultipartResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using MultipartResolver [" +
this.multipartResolver + "]");
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        // Default is no multipart resolver.
        this.multipartResolver = null;
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate MultipartResolver with name '"
+
            MULTIPART_RESOLVER_BEAN_NAME +
            "': no multipart request handling provided");
        }
    }
}
```

因为之前的步骤已经完成了Spring中配置文件的解析，所以在这里只要在配置文件注册过都可以通过 ApplicationContext 提供的 getBean 方法来直接获取对应 bean，进而初始化MultipartResolver中的multipartResolver变量。

（2）初始化LocaleResolver。

在Spring的国际化配置中一共有3种使用方式。

基于URL参数的配置。

通过URL参数来控制国际化，比如你在页面上加一句简体中文来控制项目中使用的国际化参数。而提供这个功能的就是AcceptHeaderLocaleResolver，默认的参数名为locale，注意大小写。里面放的就是你的提交参数，比如en_US、zh_CN之类的，具体配置如下；

<bean id="localeResolver"
class="org.Springframework.web.servlet.i18n. AcceptHeader
LocaleResolver"/>

基于session的配置。

它通过检验用户会话中预置的属性来解析区域。最常用的是根据用户本次会话过程中的语言设定决定语言种类（例如，用户登录时选择语言种类，则此次登录周期内统一使用此语言设定），如果该会话属性不存在，它会根据 accept-language HTTP头部确定默认区域。

<bean id="localeResolver"
class="org.Springframework.web.servlet.i18n.SessionLocaleResolver"/>

基于Cookie的国际化配置。

CookieLocaleResolver用于通过浏览器的cookie设置取得Locale对象。这种策略在应用程序不支持会话或者状态必须保存在客户端时有用，配置如下：

<bean id="localeResolver"
class="org.Springframework.web.servlet.i18n.CookieLocaleResolver"/>

这 3 种方式都可以解决国际化的问题，但是，对于 LocalResolver 的使用基础是在DispatcherServlet中的初始化。

```
private void initLocaleResolver(ApplicationContext context) {
    try {
```

```
            this.localeResolver =
context.getBean(LOCALE_RESOLVER_BEAN_NAME, Locale确
            良Resolver.class);
            if (logger.isDebugEnabled()) {
                logger.debug("Using LocaleResolver [" + this.localeResolver +
"]");
            }
        }
        catch (NoSuchBeanDefinitionException ex) {
            // We need to use the default.
            this.localeResolver = getDefaultStrategy(context,
LocaleResolver.class);
            if (logger.isDebugEnabled()) {
                logger.debug("Unable to locate LocaleResolver with name '" +
                LOCALE_RESOLVER_BEAN_NAME +
                "': using default [" + this.localeResolver + "]");
            }
        }
    }
```

提取配置文件中设置的LocaleResolver来初始化DispatcherServlet中的localeResolver属性。

（3）初始化ThemeResolver。

在Web开发中经常会遇到通过主题Theme来控制网页风格，这将进一步改善用户体验。简单地说，一个主题就是一组静态资源（比如样式表和图片），它们可以影响应用程序的视觉效果。Spring中的主题功能和国际化功能非常类似。构成Spring主题功能主要包括如下内容。

主题资源。

org.Springframework.ui.context.ThemeSource是Spring中主题资源的接口，Spring的主题需要通过ThemeSource接口来实现存放主题信息的资源。

org.Springframework.ui.context.support.ResourceBundleThemeSource是ThemeSource接口默认实现类（也就是通过ResourceBundle资源的方式定义主题），在Spring中的配置如下：

```
<bean id="themeSource"
class="org.Springframework.ui.context.support.ResourceBundle
    ThemeSource">
    <property name="basenamePrefix" value="com.test. "></property>
    </bean>
```

默认状态下是在类路径根目录下查找相应的资源文件，也可以通过basenamePrefix来制定。这样，DispatcherServlet就会在com.test包下查找资源文件。

主题解析器。

ThemeSource 定义了一些主题资源，那么不同的用户使用什么主题资源由谁定义呢？org.Springframework.web.servlet.ThemeResolver是主题解析器的接口，主题解析的工作便是由它的子类来完成。

对于主题解析器的子类主要有3个比较常用的实现。以主题文件summer.properties为例。

n FixedThemeResolver用于选择一个固定的主题。

```
<bean id="themeResolver"
class="org.Springframework.web.servlet.theme.FixedTheme Resolver">
    <property name="defaultThemeName" value="summer"/>
    </bean>
```

以上配置的作用是设置主题文件为summer.properties，在整个项目内固定不变。

o CookieThemeResolver用于实现用户所选的主题，以cookie的形式存放在客户端的机器上，配置如下：

<bean id="themeResolver"
class="org.Springframework.web.servlet.theme.CookieThemeResolver">

<property name="defaultThemeName" value="summer"/>

</bean>

pSessionThemeResolver用于主题保存在用户的HTTP Session中。

<bean id="themeResolver"
class="org.Springframework.web.servlet.theme.SessionThemeResolver">

<property name="defaultThemeName" value="summer"/>

</bean>

以上配置用于设置主题名称，并且将该名称保存在用户的HttpSession中。

q AbstractThemeResolver是一个抽象类被SessionThemeResolver和FixedThemeResolver继承，用户也可以继承它来自定义主题解析器。

拦截器。

如果需要根据用户请求来改变主题，那么 Spring 提供了一个已经实现的拦截器 Theme ChangeInterceptor拦截器了，配置如下：

<bean id="themeChangeInterceptor"
class="org.Springframework.web.servlet.theme. Theme
ChangeInterceptor">

<property name="paramName" value="themeName"></property>

</bean>

其中设置用户请求参数名为themeName，即URL为?themeName=具体的主题名称。此外，还需要在handlerMapping中配置拦截器。当然需要在HandleMapping中添加拦截器。

<property name="interceptors">

```
        <list>
            <ref local="themeChangeInterceptor" />
        </list>
    </property>
```

了解了主题文件的简单使用方式后，再来查看解析器的初始化工作，与其他变量的初始化工作相同，主题文件解析器的初始化工作并没有任何特别需要说明的地方。

```
private void initThemeResolver(ApplicationContext context) {
    try {
        this.themeResolver =
context.getBean(THEME_RESOLVER_BEAN_NAME, ThemeResolver.
        class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using ThemeResolver [" + this.themeResolver +
"]");
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        // We need to use the default.
        this.themeResolver = getDefaultStrategy(context,
ThemeResolver.class);
        if (logger.isDebugEnabled()) {
            logger.debug(
            "Unable to locate ThemeResolver with name '" + THEME_
            RESOLVER_BEAN_NAME + "': using default [" +
            this.themeResolver + "]");
        }
```

```
    }
}
```

（4）初始化HandlerMappings。

当客户端发出 Request 时 DispatcherServlet 会将 Request 提交给 HandlerMapping，然后HanlerMapping 根据 Web Application Context 的配置来回传给 DispatcherServlet 相应的Controller。

在基于 SpringMVC 的 Web 应用程序中，我们可以为 DispatcherServlet 提供多个 Handler Mapping供其使用。DispatcherServlet 在选用HandlerMapping的过程中，将根据我们所指定的一系列 HandlerMapping 的优先级进行排序，然后优先使用优先级在前的 HandlerMapping。如果当前的HandlerMapping能够返回可用的Handler，DispatcherServlet则使用当前返回的Handler进行Web请求的处理，而不再继续询问其他的HandlerMapping。否则，DispatcherServlet将继续按照各个HandlerMapping的优先级进行询问，直到获取一个可用的Handler为止。初始化配置如下：

```
private void initHandlerMappings(ApplicationContext context) {
    this.handlerMappings = null;
    if (this.detectAllHandlerMappings) {
        Map<String, HandlerMapping> matchingBeans =
        BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
        HandlerMapping. class, true, false);
        if (!matchingBeans.isEmpty()) {
            this.handlerMappings = new ArrayList<HandlerMapping>
(matchingBeans.
            values());
            // We keep HandlerMappings in sorted order.
            OrderComparator.sort(this.handlerMappings);
```

```
            }
        }
        else {
            try {
                HandlerMapping hm =
context.getBean(HANDLER_MAPPING_BEAN_NAME,
               HandlerMapping.class);
                this.handlerMappings = Collections.singletonList(hm);
            }
            catch (NoSuchBeanDefinitionException ex) {
                // Ignore, we'll add a default HandlerMapping later.
            }
        }
        // Ensure we have at least one HandlerMapping, by registering
        // a default HandlerMapping if no other mappings are found.
        if (this.handlerMappings == null) {
            this.handlerMappings = getDefaultStrategies(context,
HandlerMapping.class);
            if (logger.isDebugEnabled()) {
                logger.debug("No HandlerMappings found in servlet '" +
getServletName()
                + "': using default");
            }
        }
    }
```

默认情况下，SpringMVC将加载当前系统中所有实现了HandlerMapping接口的bean。如果只期望SpringMVC加载指定的

handlermapping时，可以修改web.xml中的DispatcherServlet的初始参数，将detectAllHandlerMappings的值设置为false：

```
<init-param>
   <param-name>detectAllHandlerMappings</param-name>
   <param-value>false</param-value>
</init-param>
```

此时，SpringMVC将查找名为“handlerMapping”的bean，并作为当前系统中唯一的handlermapping。如果没有定义handlerMapping的话，则SpringMVC将按照org.Springframework. web.servlet.DispatcherServlet所在目录下的DispatcherServlet.properties中所定义的org.Springframework.web.servlet.HandlerMapping的内容来加载默认的handlerMapping（用户没有自定义Strategies的情况下）。

（5）初始化HandlerAdapters。

从名字也能联想到这是一个典型的适配器模式的使用，在计算机编程中，适配器模式将一个类的接口适配成用户所期待的。使用适配器，可以使接口不兼容而无法在一起工作的类协同工作，做法是将类自己的接口包裹在一个已存在的类中。那么在处理handler时为什么会使用适配器模式呢？回答这个问题我们首先要分析它的初始化逻辑。

```
private void initHandlerAdapters(ApplicationContext context) {
    this.handlerAdapters = null;
    if (this.detectAllHandlerAdapters) {
        // Find all HandlerAdapters in the ApplicationContext, including
ancestor
        contexts.
        Map<String, HandlerAdapter> matchingBeans =
        BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
        HandlerAdapter.class, true, false);
```

```
                if (!matchingBeans.isEmpty()) {
                    this.handlerAdapters = new ArrayList<HandlerAdapter>
(matchingBeans.
                   values());
                    // We keep HandlerAdapters in sorted order.
                   OrderComparator.sort(this.handlerAdapters);
                }
            }
            else {
                try {
                    HandlerAdapter ha =
context.getBean(HANDLER_ADAPTER_BEAN_NAME,
                   HandlerAdapter.class);
                    this.handlerAdapters = Collections.singletonList(ha);
                }
                catch (NoSuchBeanDefinitionException ex) {
                    // Ignore, we'll add a default HandlerAdapter later.
                }
            }
            // Ensure we have at least some HandlerAdapters, by registering
            // default HandlerAdapters if no other adapters are found.
            if (this.handlerAdapters == null) {
                this.handlerAdapters = getDefaultStrategies(context,
HandlerAdapter.class);
                if (logger.isDebugEnabled()) {
                    logger.debug("No HandlerAdapters found in servlet '" +
getServletName()
```

```
            + "': using default");
        }
    }
}
```

同样在初始化的过程中涉及了一个变量detectAllHandlerAdapters，detectAllHandlerAdapters作用和detectAllHandlerMappings类似，只不过作用对象为handlerAdapter。亦可通过如下配置来强制系统只加载bean name为“handlerAdapter”handlerAdapter。

```
<init-param>
   <param-name>detectAllHandlerAdapters</param-name>
   <param-value>false</param-value>
</init-param>
```

如果无法找到对应的bean，那么系统会尝试加载默认的适配器。

```
protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T>
strategyInterface) {
     String key = strategyInterface.getName();
     String value = defaultStrategies.getProperty(key);
     if (value != null) {
        String[] classNames =
StringUtils.commaDelimitedListToStringArray(value);
        List<T> strategies = new ArrayList<T>(classNames.length);
        for (String className : classNames) {
           try {
              Class<?> clazz = ClassUtils.forName(className, DispatcherServlet.
             class.getClassLoader());
```

```
            Object strategy = createDefaultStrategy(context, clazz);
            strategies.add((T) strategy);
        }
        catch (ClassNotFoundException ex) {
            throw new BeanInitializationException(
            "Could not find DispatcherServlet's default strategy
            class [" + className +
            "] for interface [" + key + "]", ex);
        }
        catch (LinkageError err) {
            throw new BeanInitializationException(
            "Error loading DispatcherServlet's default strategy
            class [" + className +
            "] for interface [" + key + "]: problem with
            class file or dependent class", err);
        }
    }
    return strategies;
}
else {
    return new LinkedList<T>();
}
}
```

在 getDefaultStrategies 函数中， Spring 会尝试从 defaultStrategies 中加载对应的HandlerAdapter的属性，那么defaultStrategies是如何初始化的呢？

在当前类DispatcherServlet中存在这样一段初始化代码块：

```
static {
    try {
        // DEFAULT_STRATEGIES_PATH =
"DispatcherServlet.properties";
    ClassPathResource resource = new
ClassPathResource(DEFAULT_STRATEGIES_PATH,
   DispatcherServlet.class);
        defaultStrategies =
PropertiesLoaderUtils.loadProperties(resource);
    }
    catch (IOException ex) {
        throw new IllegalStateException("Could not load 'Dispatcher
Servlet.
        properties': " + ex.getMessage());
    }
}
```

在系统加载的时候，defaultStrategies根据当前路径DispatcherServlet.properties来初始化本身，查看DispatcherServlet.properties中对应于HandlerAdapter的属性：

```
org.Springframework.web.servlet.HandlerAdapter=org.Springframework
pRequestHandlerAdapter,\

org.Springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.Springframework.web.servlet.mvc.annotation.AnnotationMethodHandler
```

由此得知，如果程序开发人员没有在配置文件中定义自己的适配器，那么Spring会默认加载配置文件中的3个适配器。

作为总控制器的派遣器servlet通过处理器映射得到处理器后，会轮询处理器适配器模块，查找能够处理当前HTTP请求的处理器适配器的实现，处理器适配器模块根据处理器映射返回的处理器类型，例如简单的控制器类型、注解控制器类型或者远程调用处理器类型，来选择某一个适当的处理器适配器的实现，从而适配当前的HTTP请求。

HTTP请求处理器适配器（HttpRequestHandlerAdapter）。

HTTP请求处理器适配器仅仅支持对HTTP请求处理器的适配。它简单地将HTTP请求对象和响应对象传递给HTTP请求处理器的实现，它并不需要返回值。它主要应用在基于HTTP的远程调用的实现上。

简单控制器处理器适配器（SimpleControllerHandlerAdapter）。

这个实现类将HTTP请求适配到一个控制器的实现进行处理。这里控制器的实现是一个简单的控制器接口的实现。简单控制器处理器适配器被设计成一个框架类的实现，不需要被改写，客户化的业务逻辑通常是在控制器接口的实现类中实现的。

注解方法处理器适配器（AnnotationMethodHandlerAdapter）。

这个类的实现是基于注解的实现，它需要结合注解方法映射和注解方法处理器协同工作。它通过解析声明在注解控制器的请求映射信息来解析相应的处理器方法来处理当前的HTTP请求。在处理的过程中，它通过反射来发现探测处理器方法的参数，调用处理器方法，并且映射返回值到模型和控制器对象，最后返回模型和控制器对象给作为主控制器的派遣器Servlet。

所以我们现在基本上可以回答之前的问题了，Spring中所使用的Handler并没有任何特殊的联系，但是为了统一处理，Spring提供了不同情况下的适配器。

（6）初始化HandlerExceptionResolvers。

基于HandlerExceptionResolver接口的异常处理，使用这种方式只需要实现resolveException方法，该方法返回一个ModelAndView对象，在

方法内部对异常的类型进行判断，然后尝试生成对应的 ModelAndView 对象，如果该方法返回了 null，则 Spring 会继续寻找其他的实现了 HandlerExceptionResolver接口的bean。换句话说，Spring会搜索所有注册在其环境中的实现了HandlerExceptionResolver接口的bean，逐个执行，直到返回了一个ModelAndView对象。

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.Springframework.stereotype.Component;
import org.Springframework.web.servlet.HandlerExceptionResolver;
import org.Springframework.web.servlet.ModelAndView;
@Component
public class ExceptionHandler implements HandlerExceptionResolver
{
    private static final Log logs =
LogFactory.getLog(ExceptionHandler.class);
   @Override
    public ModelAndView resolveException(HttpServletRequest
request, HttpServletResponse
    response, Object obj,
       Exception exception)
   {
       request.setAttribute("exception", exception.toString());
       request.setAttribute("exceptionStack", exception);
       logs.error(exception.toString(), exception);
       return new ModelAndView("error/exception");
```

```
    }
}
```

这个类必须声明到Spring中去，让Spring管理它，在Spring的配置文件applicationContext.xml中增加以下内容：

```
<bean id="exceptionHandler"
class="com.test.exception.MyExceptionHandler"/>
```

初始化代码如下：

```
private void initHandlerExceptionResolvers(ApplicationContext
context) {
    this.handlerExceptionResolvers = null;
    if (this.detectAllHandlerExceptionResolvers) {
        // Find all HandlerExceptionResolvers in the ApplicationContext,
including
        ancestor contexts.
        Map<String, HandlerExceptionResolver> matchingBeans =
BeanFactoryUtils
            .beansOfTypeIncludingAncestors(context,
HandlerExceptionResolver.
            class, true, false);
        if (!matchingBeans.isEmpty()) {
            this.handlerExceptionResolvers = new ArrayList<Handler
Exception
           Resolver>(matchingBeans.values());
            // We keep HandlerExceptionResolvers in sorted order.
           OrderComparator.sort(this.handlerExceptionResolvers);
        }
    }
```

```
            else {
                try {
                    HandlerExceptionResolver her =

context.getBean(HANDLER_EXCEPTION_RESOLVER_BEAN_NAME,
                   HandlerExceptionResolver.class);
                    this.handlerExceptionResolvers =
Collections.singletonList(her);
                }
                catch (NoSuchBeanDefinitionException ex) {
                    // Ignore, no HandlerExceptionResolver is fine too.
                }
            }
            // Ensure we have at least some HandlerExceptionResolvers, by
registering
            // default HandlerExceptionResolvers if no other resolvers are found.
            if (this.handlerExceptionResolvers == null) {
                this.handlerExceptionResolvers = getDefaultStrategies(context,
Handler
               ExceptionResolver.class);
                if (logger.isDebugEnabled()) {
                    logger.debug("No HandlerExceptionResolvers found in servlet
'" +
                    getServletName() + "': using default");
                }
            }
        }
```

（7）初始化RequestToViewNameTranslator。

当Controller处理器方法没有返回一个View对象或逻辑视图名称，并且在该方法中没有直接往response的输出流里面写数据的时候，Spring就会采用约定好的方式提供一个逻辑视图名称。这个逻辑视图名称是通过 Spring 定义的 org.Springframework.web.servlet.RequestToView NameTranslator接口的 getViewName方法来实现的，我们可以实现自己的Request ToViewName Translator接口来约定好没有返回视图名称的时候如何确定视图名称。Spring已经给我们提供了一个它自己的实现，那就是 org.Springframework.web.servlet.view.DefaultRequest ToViewName Translator。

在介绍 DefaultRequestToViewNameTranslator 是如何约定视图名称之前，先来看一下它支持用户定义的属性。

prefix：前缀，表示约定好的视图名称需要加上的前缀，默认是空串。

suffix：后缀，表示约定好的视图名称需要加上的后缀，默认是空串。

separator：分隔符，默认是斜杠“/”。

stripLeadingSlash：如果首字符是分隔符，是否要去除，默认是true。

stripTrailingSlash：如果最后一个字符是分隔符，是否要去除，默认是true。

stripExtension：如果请求路径包含扩展名是否要去除，默认是true。

urlDecode：是否需要对 URL 解码，默认是 true。它会采用 request指定的编码或者ISO-8859-1编码对URL进行解码。

当我们没有在 SpringMVC 的配置文件中手动的定义一个名为viewNameTranlator 的 Bean的时候，Spring就会为我们提供一个默认的

viewNameTranslator，即DefaultRequest ToViewName Translator。

接下来看一下，当 Controller 处理器方法没有返回逻辑视图名称时，DefaultRequestToView NameTranslator 是如何约定视图名称的。DefaultRequestToViewNameTranslator 会获取到请求的URI，然后根据提供的属性做一些改造，把改造之后的结果作为视图名称返回。这里以请求路径http://localhost/app/test/index.html为例，来说明一下DefaultRequestToViewNameTranslator是如何工作的。该请求路径对应的请求 URI 为/test/index.html，我们来看以下几种情况，它分别对应的逻辑视图名称是什么。

prefix 和 suffix 如果都存在，其他为默认值，那么对应返回的逻辑视图名称应该是prefixtest/indexsuffix。

stripLeadingSlash和stripExtension都为false，其他默认，这时候对应的逻辑视图名称是/product/index.html。

都采用默认配置时，返回的逻辑视图名称应该是product/index。

如果逻辑视图名称跟请求路径相同或者相关关系都是一样的，那么我们就可以采用Spring为我们事先约定好的逻辑视图名称返回，这可以大大简化我们的开发工作，而以上功能实现的关键属性viewNameTranslator，则是在initRequestToViewNameTranslator中完成。

```
private void initRequestToViewNameTranslator(ApplicationContext
context) {
        try {
           this.viewNameTranslator =

context.getBean(REQUEST_TO_VIEW_NAME_TRANSLATOR_BEAN_N
          RequestToViewNameTranslator.class);
           if (logger.isDebugEnabled()) {
              logger.debug("Using RequestToViewNameTranslator [" +
```

```
this.viewName
                Translator + "]");
            }
        }
        catch (NoSuchBeanDefinitionException ex) {
            // We need to use the default.
            this.viewNameTranslator = getDefaultStrategy(context,
RequestToViewName
            Translator.class);
            if (logger.isDebugEnabled()) {
                logger.debug("Unable to locate RequestToViewNameTranslator
with name '" +

REQUEST_TO_VIEW_NAME_TRANSLATOR_BEAN_NAME + "': using
default
                [" + this.viewNameTranslator +
               "]");
            }
        }
    }
```

（8） 初始化ViewResolvers。

在 SpringMVC 中，当 Controller 将请求处理结果放入到 ModelAndView 中以后， DispatcherServlet会根据ModelAndView选择合适的视图进行渲染。那么在SpringMVC中是如何选择合适的View呢？View对象是是如何创建的呢？答案就在ViewResolver中。ViewResolver接口定义了resolverViewName方法，根据viewName创建合适类型的View实现。

那么如何配置ViewResolver呢？在Spring中，ViewResolver作为Spring Bean存在，可以在Spring配置文件中进行配置，例如下面的代码，配置了JSP相关的viewResolver。

```
<bean
class="org.Springframework.web.servlet.view.InternalResourceViewResolver
    <property name="prefix" value="/WEB-INF/views/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

viewResolvers属性的初始化工作在initViewResolvers中完成。

```
private void initViewResolvers(ApplicationContext context) {
    this.viewResolvers = null;
    if (this.detectAllViewResolvers) {
        // Find all ViewResolvers in the ApplicationContext, including
ancestor
        contexts.
        Map<String, ViewResolver> matchingBeans =
            BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
            ViewResolver.class, true, false);
        if (!matchingBeans.isEmpty()) {
            this.viewResolvers = new ArrayList<ViewResolver>
(matchingBeans.
            values());
            // We keep ViewResolvers in sorted order.
            OrderComparator.sort(this.viewResolvers);
        }
    }
    else {
```

```
            try {
                ViewResolver vr =
context.getBean(VIEW_RESOLVER_BEAN_NAME, ViewResolver.
               class);
                this.viewResolvers = Collections.singletonList(vr);
            }
            catch (NoSuchBeanDefinitionException ex) {
                // Ignore, we'll add a default ViewResolver later.
            }
        }
        // Ensure we have at least one ViewResolver, by registering
        // a default ViewResolver if no other resolvers are found.
        if (this.viewResolvers == null) {
            this.viewResolvers = getDefaultStrategies(context,
ViewResolver.class);
            if (logger.isDebugEnabled()) {
                logger.debug("No ViewResolvers found in servlet '" +
getServletName()
                + "': using default");
            }
        }
    }
```

（9）初始化FlashMapManager。

SpringMVC Flash attributes提供了一个请求存储属性，可供其他请求使用。在使用重定向时候非常必要，例如Post/Redirect/Get模式。Flash attributes在重定向之前暂存（就像存在 session中）以便重定向之后还能使用，并立即删除。

SpringMVC有两个主要的抽象来支持 flash attributes。FlashMap用于保持 flash attributes，而 FlashMapManager用于存储、检索、管理FlashMap实例。

flash attribute支持默认开启（"on"）并不需要显式启用，它永远不会导致HTTP Session的创建。这两个FlashMap实例都可以通过静态方法RequestContextUtils从Spring MVC的任何位置访问。

flashMapManager的初始化在initFlashMapManager中完成。

```
private void initFlashMapManager(ApplicationContext context) {
    try {
        this.flashMapManager =
        context.getBean(FLASH_MAP_MANAGER_BEAN_NAME,
FlashMapManager.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using FlashMapManager [" +
this.flashMapManager + "]");
        }
    }
    catch (NoSuchBeanDefinitionException ex) {
        // We need to use the default.
        this.flashMapManager = getDefaultStrategy(context,
FlashMapManager.
        class);
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate FlashMapManager with name '"
+
            FLASH_MAP_MANAGER_BEAN_NAME + "': using default
[" +
```

```
            this.flashMapManager + "]");
        }
    }
}
```

11.4 DispatcherServlet的逻辑处理

根据之前的示例，我们知道在 HttpServlet 类中分别提供了相应的服务方法，它们是doDelete()、doGet()、doOptions()、doPost()、doPut()和doTrace()，它会根据请求的不同形式将程序引导至对应的函数进行处理。这几个函数中最常用的函数无非就是doGet()和doPost()，那么我们就直接查看DispatcherServlet中对于这两个函数的逻辑实现。

```
@Override
protected final void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
@Override
protected final void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
```

对于不同的方法，Spring 并没有做特殊处理，而是统一将程序再一次地引导至 process Request(request, response)中。

```
protected final void processRequest(HttpServletRequest request,
```

```
HttpServletResponse
        response)
            throws ServletException, IOException {
        //记录当前时间，用于计算web请求的处理时间
            long startTime = System.currentTimeMillis();
            Throwable failureCause = null;
            // Expose current LocaleResolver and request as LocaleContext.
            LocaleContext previousLocaleContext =
LocaleContextHolder.getLocaleContext();
            LocaleContextHolder.setLocaleContext(buildLocaleContext(request),
            this.threadContextInheritable);
            // Expose current RequestAttributes to current thread.
            RequestAttributes previousRequestAttributes =
RequestContextHolder. GetRequest
            Attributes();
            ServletRequestAttributes requestAttributes = null;
            if (previousRequestAttributes == null || previousRequestAttributes.
getClass().
            equals(ServletRequestAttributes.class)) {
                requestAttributes = new ServletRequestAttributes(request);
               RequestContextHolder.setRequestAttributes(requestAttributes,
               this.threadContextInheritable);
            }
            if (logger.isTraceEnabled()) {
                logger.trace("Bound request context to thread: " + request);
            }
            try {
```

```
                doService(request, response);
            }
            catch (ServletException ex) {
                failureCause = ex;
                throw ex;
            }
            catch (IOException ex) {
                failureCause = ex;
                throw ex;
            }
            catch (Throwable ex) {
                failureCause = ex;
                throw new NestedServletException("Request processing failed",
ex);
            }
            finally {
                // Clear request attributes and reset thread-bound context.

LocaleContextHolder.setLocaleContext(previousLocaleContext,this.threadCo
                Inheritable);
                if (requestAttributes != null) {

RequestContextHolder.setRequestAttributes(previousRequestAttributes,
                    this.threadContextInheritable);
                    requestAttributes.requestCompleted();
                }
                if (logger.isTraceEnabled()) {
```

```
                    logger.trace("Cleared thread-bound request context: " +
request);
                }
                if (logger.isDebugEnabled()) {
                    if (failureCause != null) {
                        this.logger.debug("Could not complete request",
failureCause);
                    }
                    else {
                        this.logger.debug("Successfully completed request");
                    }
                }
                if (this.publishEvents) {
                    // Whether or not we succeeded, publish an event.
                    long processingTime = System.currentTimeMillis() - startTime;
                   this.webApplicationContext.publishEvent(
                    new ServletRequestHandledEvent(this,
                    request.getRequestURI(), request.getRemoteAddr(),
                   request.getMethod(),
                   getServletConfig().getServletName(),
                   WebUtils.getSessionId(request),
                   getUsernameForRequest(request),
                    processingTime, failureCause));
                }
            }
        }
```

函数中已经开始了对请求的处理，虽然把细节转移到了 doService

函数中实现，但是我们不难看出处理请求前后所做的准备与处理工作。

（1）为了保证当前线程的LocaleContext以及RequestAttributes可以在当前请求后还能恢复，提取当前线程的两个属性。

（2）根据当前request创建对应的LocaleContext和RequestAttributes，并绑定到当前线程。

（3）委托给doService方法进一步处理。

（4）请求处理结束后恢复线程到原始状态。

（5）请求处理结束后无论成功与否发布事件通知。

继续查看doService方法。

```
protected void doService(HttpServletRequest request,
HttpServletResponse response) throws
    Exception {
        if (logger.isDebugEnabled()) {
            String requestUri = urlPathHelper.getRequestUri(request);
            logger.debug("DispatcherServlet with name '" + getServletName()
+ "'
            processing " + request.getMethod() +
            " request for [" + requestUri + "]");
        }
        // Keep a snapshot of the request attributes in case of an include,
        // to be able to restore the original attributes after the include.
        Map<String, Object> attributesSnapshot = null;
        if (WebUtils.isIncludeRequest(request)) {
            logger.debug("Taking snapshot of request attributes before
include");
            attributesSnapshot = new HashMap<String, Object>();
            Enumeration<?> attrNames = request.getAttributeNames();
```

```
            while (attrNames.hasMoreElements()) {
                String attrName = (String) attrNames.nextElement();
                if (this.cleanupAfterInclude || attrName.startsWith
("org.Springframework.
                web.servlet")) {
                    attributesSnapshot.put(attrName, request.getAttribute
(attrName));
                }
            }
        }
        // Make framework objects available to handlers and view objects.

request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
getWebApplicationContext());
        request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE,
this.localeResolver);
        request.setAttribute(THEME_RESOLVER_ATTRIBUTE,
this.themeResolver);
        request.setAttribute(THEME_SOURCE_ATTRIBUTE,
getThemeSource());
        FlashMap inputFlashMap =
this.flashMapManager.retrieveAndUpdate(request,
        response);
        if (inputFlashMap != null) {
            request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE,
Collections.unmodifiableMap
            (inputFlashMap));
```

```
        }
         request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new
FlashMap());
         request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE,
this.flashMapManager);
         try {
            doDispatch(request, response);
        }
         finally {
            // Restore the original attribute snapshot, in case of an include.
            if (attributesSnapshot != null) {
               restoreAttributesAfterInclude(request, attributesSnapshot);
            }
        }
    }
```

我们猜想对请求处理至少应该包括一些诸如寻找Handler并页面跳转之类的逻辑处理，但是，在 doService 中我们并没有看到想看到的逻辑，相反却同样是一些准备工作，但是这些准备工作却是必不可少的。Spring 将已经初始化的功能辅助工具变量，比如 localeResolver、themeResolver等设置在request属性中，而这些属性会在接下来的处理中派上用场。

经过层层的准备工作，终于在doDispatch函数中看到了完整的请求处理过程。

```
    protected void doDispatch(HttpServletRequest request,
HttpServletResponse response)
    throws Exception {
         HttpServletRequest processedRequest = request;
```

```
HandlerExecutionChain mappedHandler = null;
int interceptorIndex = -1;
try {
    ModelAndView mv;
    boolean errorView = false;
    try {
//如果是MultipartContent类型的request则转换request为
MultipartHttpServletRequest 类型的
request
    processedRequest = checkMultipart(request);
//根据request信息寻找对应的Handler
    mappedHandler = getHandler(processedRequest, false);
    if (mappedHandler == null || mappedHandler.getHandler() == null)
{
        //如果没有找到对应的handler则通过response反馈错误信息
        noHandlerFound(processedRequest, response);
        return;
    }
    //根据当前的handler寻找对应的HandlerAdapter
    HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());
//如果当前handler支持last-modified头处理
    String method = request.getMethod();
    boolean isGet = "GET".equals(method);
    if (isGet || "HEAD".equals(method)) {
        long lastModified = ha.getLastModified(request,
mappedHandler.
```

```
getHandler());
    if (logger.isDebugEnabled()) {
        String requestUri = urlPathHelper.getRequestUri(request);
        logger.debug("Last-Modified value for [" + requestUri + "] is:
        " + lastModified);
    }
    if (new ServletWebRequest(request,
response).checkNotModified
    (lastModified) && isGet) {
       return;
    }
}
   //拦截器的preHandler方法的调用
    HandlerInterceptor[] interceptors =
mappedHandler.getInterceptors();
    if (interceptors != null) {
        for (int i = 0; i < interceptors.length; i++) {
            HandlerInterceptor interceptor = interceptors[i];
            if (!interceptor.preHandle(processedRequest, response,
            mappedHandler.getHandler())) {
               triggerAfterCompletion(mappedHandler,
interceptorIndex,
                processedRequest, response, null);
               return;
           }
            interceptorIndex = i;
       }
```

```
        }
      //真正的激活handler并返回视图
        mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
        //视图名称转换应用于需要添加前缀后缀的情况
        if (mv != null && !mv.hasView()) {
          mv.setViewName(getDefaultViewName(request));
        }
        //应用所有拦截器的postHandle方法
        if (interceptors != null) {
          for (int i = interceptors.length - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
          interceptor.postHandle(processedRequest, response,
mappedHandler.
          getHandler(), mv);
          }
        }
      }
      catch (ModelAndViewDefiningException ex) {
        logger.debug("ModelAndViewDefiningException encountered",
ex);
        mv = ex.getModelAndView();
      }
      catch (Exception ex) {
        Object handler = (mappedHandler != null ?
mappedHandler.getHandler() : null);
        mv =processHandlerException(processedRequest, response,
```

```
handler, ex);
                    errorView = (mv != null);
                }
                 // Did the handler return a view to render?
                //如果在Handler实例的处理中返回了view，那么需要做页面的
处理
                 if (mv != null && !mv.wasCleared()) {
                    //处理页面跳转
                    render(mv, processedRequest, response);
                     if (errorView) {
                        WebUtils.clearErrorRequestAttributes(request);
                    }
                }
                 else {
                     if (logger.isDebugEnabled()) {
                         logger.debug("Null ModelAndView returned to
DispatcherServlet
                         with name '" + getServletName() +
                         "': assuming HandlerAdapter completed request handling");
                    }
                }
                //完成处理激活触发器
                triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest,
                 response, null);
            }
             catch (Exception ex) {
```

```
                // Trigger after-completion for thrown exception.
                triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest,
                response, ex);
                throw ex;
            }
            catch (Error err) {
                ServletException ex = new NestedServletException("Handler
processing
                failed", err);
                // Trigger after-completion for thrown exception.
                triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest,
                response, ex);
                throw ex;
            }
            finally {
                // Clean up any resources used by a multipart request.
                if (processedRequest != request) {
                   cleanupMultipart(processedRequest);
                }
            }
        }
```

doDispatch 函数中展示了 Spring 请求处理所涉及的主要逻辑，而我们之前设置在 request中的各种辅助属性也都有被派上了用场。下面回顾一下逻辑处理的全过程。

11.4.1 MultipartContent类型的 request处理

对于请求的处理，Spring首先考虑的是对于Multipart的处理，如果是MultipartContent类型的request，则转换request为MultipartHttpServletRequest 类型的request。

```
protected HttpServletRequest checkMultipart(HttpServletRequest
request) throws
    MultipartException {
        if (this.multipartResolver != null &&
this.multipartResolver.isMultipart(request)) {
            if (request instanceof MultipartHttpServletRequest) {
                logger.debug("Request is already a MultipartHttpServletRequest
- if not
                in a forward, " +
                "this typically results from an additional MultipartFilter in
web.xml");
            }
            else {
                return this.multipartResolver.resolveMultipart(request);
            }
        }
        // If not returned before: return original request.
        return request;
}
```

11.4.2 根据 request信息寻找对应的Handler

在Spring中最简单的映射处理器配置如下：

```
<bean id="simpleUrlMapping"
```

```
class="org.Springframework.web.servlet.handler.SimpleUrlHandlerMapping"
        <property name="mappings">
          <props>
              <prop key="/userlist.htm">userController</prop>
          </props>
        </property>
    </bean>
```

在 Spring 加载的过程中，Spring 会将类型为 SimpleUrlHandlerMapping 的实例加载到this.handlerMappings中，按照常理推断，根据request提取对应的Handler，无非就是提取当前实例中的userController，但是userController为继承自AbstractController类型实例，与Handler ExecutionChain并无任何关联，那么这一步是如何封装的呢？

```
    protected HandlerExecutionChain getHandler(HttpServletRequest request, boolean cache)
    throws Exception {
       return getHandler(request);
    }
    protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
         for (HandlerMapping hm : this.handlerMappings) {
            if (logger.isTraceEnabled()) {
               logger.trace(
                "Testing handler map [" + hm + "] in DispatcherServlet with
                name '" + getServletName() + "'");
            }
            HandlerExecutionChain handler = hm.getHandler(request);
```

```
            if (handler != null) {
                return handler;
            }
        }
        return null;
    }
```

在之前的内容我们提过，在系统启动时 Spring 会将所有的映射类型的 bean 注册到this.handlerMappings 变量中，所以此函数的目的就是遍历所有的 HandlerMapping，并调用其getHandler方法进行封装处理。以SimpleUrlHandlerMapping为例查看其getHandler方法如下：

```
    public final HandlerExecutionChain getHandler(HttpServletRequest
request) throws
    Exception {
        //根据request获取对应的handler
        Object handler = getHandlerInternal(request);
        if (handler == null) {
            //如果没有对应request的handler则使用默认的handler
            handler = getDefaultHandler();
        }
    //如果也没有提供默认的handler则无法继续处理返回null
        if (handler == null) {
            return null;
        }
        // Bean name or resolved handler?
        if (handler instanceof String) {
            String handlerName = (String) handler;
            handler = getApplicationContext().getBean(handlerName);
```

}

return getHandlerExecutionChain(handler, request);

}

函数中首先会使用 getHandlerInternal 方法根据 request 信息获取对应的 Handler，如果以SimpleUrlHandlerMapping为例分析，那么我们推断此步骤提供的功能很可能就是根据URL找到匹配的Controller并返回，当然如果没有找到对应的Controller 处理器那么程序会尝试去查找配置中的默认处理器，当然，当查找的controller为String类型时，那就意味着返回的是配置的bean名称，需要根据bean名称查找对应的bean，最后，还要通过getHandlerExecutionChain方法对返回的Handler进行封装，以保证满足返回类型的匹配。下面详细分析这个过程。

1．根据request查找对应的Handler

首先从根据request查找对应的Handler开始分析。

protected Object getHandlerInternal(HttpServletRequest request) throws Exception {

//截取用于匹配的url有效路径

String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);

//根据路径寻找Handler

Object handler = lookupHandler(lookupPath, request);

if (handler == null) {

Object rawHandler = null;

if ("/".equals(lookupPath)) {

//如果请求的路径仅仅是”/”，那么使用RootHandler进行处理

rawHandler = getRootHandler();

}

```
            if (rawHandler == null) {
                //无法找到handler则使用默认handler
                rawHandler = getDefaultHandler();
            }
            if (rawHandler != null) {
                //根据beanName获取对应的bean
                if (rawHandler instanceof String) {
                    String handlerName = (String) rawHandler;
                    rawHandler =
getApplicationContext().getBean(handlerName);
                }
                //模版方法
                validateHandler(rawHandler, request);
                handler = buildPathExposingHandler(rawHandler,
lookupPath, lookupPath, null);
            }
        }
        if (handler != null && logger.isDebugEnabled()) {
            logger.debug("Mapping [" + lookupPath + "] to " + handler);
        }
        else if (handler == null && logger.isTraceEnabled()) {
            logger.trace("No handler mapping found for [" + lookupPath +
"]");
        }
        return handler;
    }
    protected Object lookupHandler(String urlPath, HttpServletRequest
```

```
request) throws Exception {
        //直接匹配情况的处理
            Object handler = this.handlerMap.get(urlPath);
            if (handler != null) {
                // Bean name
                if (handler instanceof String) {
                    String handlerName = (String) handler;
                    handler = getApplicationContext().getBean(handlerName);
                }
                validateHandler(handler, request);
                return buildPathExposingHandler(handler, urlPath, urlPath,
null);
            }
            // 通配符匹配的处理
            List<String> matchingPatterns = new ArrayList<String>();
            for (String registeredPattern : this.handlerMap.keySet()) {
                if (getPathMatcher().match(registeredPattern, urlPath)) {
                    matchingPatterns.add(registeredPattern);
                }
            }
            String bestPatternMatch = null;
            Comparator<String> patternComparator =
getPathMatcher().getPatternComparator(urlPath);
            if (!matchingPatterns.isEmpty()) {
                Collections.sort(matchingPatterns, patternComparator);
                if (logger.isDebugEnabled()) {
                    logger.debug("Matching patterns for request [" + urlPath + "]
```

```
are " +
                    matchingPatterns);
                }
                bestPatternMatch = matchingPatterns.get(0);
            }
            if (bestPatternMatch != null) {
                handler = this.handlerMap.get(bestPatternMatch);
                // Bean name or resolved handler?
                if (handler instanceof String) {
                    String handlerName = (String) handler;
                    handler = getApplicationContext().getBean(handlerName);
                }
                validateHandler(handler, request);
                String pathWithinMapping =
getPathMatcher().extractPathWithinPattern
                (bestPatternMatch, urlPath);
                // There might be multiple 'best patterns', let's make sure we
have the correct
                URI template variables
                // for all of them
            Map<String, String> uriTemplateVariables = new
LinkedHashMap<String,
            String>();
            for (String matchingPattern : matchingPatterns) {
                if (patternComparator.compare(bestPatternMatch,
matchingPattern) == 0) {
                    uriTemplateVariables
```

```
                .putAll(getPathMatcher().extractUriTemplateVariables
                 (matchingPattern, urlPath));
              }
            }
            if (logger.isDebugEnabled()) {
                logger.debug("URI Template variables for request [" + urlPath +
"] are
                " + uriTemplateVariables);
            }
            returnbuildPathExposingHandler(handler, bestPatternMatch,
pathWithinMapping,
           uriTemplateVariables);
        }
        // No handler found...
        return null;
    }
```

根据URL获取对应Handler的匹配规则代码实现起来虽然很长，但是并不难理解，考虑了直接匹配与通配符两种情况。其中要提及的是buildPathExposingHandler函数，它将Handler封装成了HandlerExecutionChain类型。

```
    protected Object buildPathExposingHandler(Object rawHandler, String
bestMatchingPattern,
    String pathWithinMapping, Map<String, String> uriTemplateVariables)
{
        HandlerExecutionChain chain = new
HandlerExecutionChain(rawHandler);
        chain.addInterceptor(new
```

```
PathExposingHandlerInterceptor(bestMatchingPattern,
            pathWithinMapping));
             if (!CollectionUtils.isEmpty(uriTemplateVariables)) {
                 chain.addInterceptor(new
UriTemplateVariablesHandlerInterceptor (uri
                TemplateVariables));
            }
             return chain;
        }
```

在函数中我们看到了通过将Handler以参数形式传入，并构建HandlerExecutionChain类型实例，加入了两个拦截器。此时我们似乎已经了解了Spring这样大番周折的目的。链处理机制，是Spring中非常常用的处理方式，是AOP中的重要组成部分，可以方便地对目标对象进行扩展及拦截，这是非常优秀的设计。

2．加入拦截器到执行链

getHandlerExecutionChain函数最主要的目的是将配置中的对应拦截器加入到执行链中，以保证这些拦截器可以有效地作用于目标对象。

```
        protected HandlerExecutionChain getHandlerExecutionChain(Object
handler, HttpServletRequest
        request) {
             HandlerExecutionChain chain =
             (handler instanceof HandlerExecutionChain) ?
             (HandlerExecutionChain) handler : new
HandlerExecutionChain(handler);
            chain.addInterceptors(getAdaptedInterceptors());
             String lookupPath =
urlPathHelper.getLookupPathForRequest(request);
```

```
    for (MappedInterceptor mappedInterceptor : mappedInterceptors) {
        if (mappedInterceptor.matches(lookupPath, pathMatcher)) {
            chain.addInterceptor(mappedInterceptor.getInterceptor());
        }
    }
    return chain;
}
```

11.4.3 没找到对应的Handler的错误处理

每个请求都应该对应着一 Handler，因为每个请求都会在后台有相应的逻辑对应，而逻辑的实现就是在Handler中，所以一旦遇到没有找到Handler的情况（正常情况下如果没有URL匹配的Handler，开发人员可以设置默认的Handler来处理请求，但是如果默认请求也未设置就会出现Handler为空的情况），就只能通过response向用户返回错误信息。

```
protected void noHandlerFound(HttpServletRequest request,
HttpServletResponse response)
throws Exception {
    if (pageNotFoundLogger.isWarnEnabled()) {
        String requestUri = urlPathHelper.getRequestUri(request);
        pageNotFoundLogger.warn("No mapping found for HTTP request
with URI [" +
        requestUri +
        "] in DispatcherServlet with name '" + getServletName() + "'");
    }
    response.sendError(HttpServletResponse.SC_NOT_FOUND);
}
```

11.4.4 根据当前Handler寻找对应的HandlerAdapter

在WebApplicationContext的初始化过程中我们讨论了HandlerAdapters的初始化，了解了在默认情况下普通的 Web 请求会交给 SimpleControllerHandlerAdapter 去处理。下面我们以SimpleControllerHandlerAdapter为例来分析获取适配器的逻辑。

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    for (HandlerAdapter ha : this.handlerAdapters) {
        if (logger.isTraceEnabled()) {
            logger.trace("Testing handler adapter [" + ha + "]");
        }
        if (ha.supports(handler)) {
            return ha;
        }
    }
    throw new ServletException("No adapter for handler [" + handler +
    "]: Does your handler implement a supported interface like
    Controller?");
}
```

通过上面的函数我们了解到，对于获取适配器的逻辑无非就是遍历所有适配器来选择合适的适配器并返回它，而某个适配器是否适用于当前的Handler逻辑被封装在具体的适配器中。进一步查看SimpleControllerHandlerAdapter中的supports方法。

```
public boolean supports(Object handler) {
    return (handler instanceof Controller);
}
```

分析到这里，一切已经明了，SimpleControllerHandlerAdapter 就是用于处理普通的 Web请求的，而且对于SpringMVC来说，我们会把逻辑

封装至Controller的子类中，例如我们之前的引导示例 UserController 就是继承自 AbstractController，而 AbstractController 实现Controller接口。

11.4.5 缓存处理

在研究Spring对缓存处理的功能支持前，我们先了解一个概念：Last-Modified缓存机制。

（1）在客户端第一次输入URL时，服务器端会返回内容和状态码200，表示请求成功，同

时会添加一个“Last-Modified”的响应头，表示此文件在服务器上的最后更新时间，例如，“Last-Modified:Wed, 14 Mar 2012 10:22:42 GMT”表示最后更新时间为（2012-03-14 10:22）。

（2）客户端第二次请求此URL时，客户端会向服务器发送请求头“If-Modified-Since”，询问服务器该时间之后当前请求内容是否有被修改过，如“If-Modified-Since: Wed, 14 Mar 2012 10:22:42 GMT”，如果服务器端的内容没有变化，则自动返回HTTP 304状态码（只要响应头，内容为空，这样就节省了网络带宽）。

Spring提供的对Last-Modified机制的支持，只需要实现LastModified接口，如下所示：

```
public class HelloWorldLastModifiedCacheController extends AbstractController implements
LastModified {
    private long lastModified;
    protected ModelAndView handleRequestInternal(HttpServletRequest req, Http
    ServletResponse resp) throws Exception {
      //点击后再次请求当前页面
      resp.getWriter().write("<a href=">this</a>");
```

```
        return null;
    }
    public long getLastModified(HttpServletRequest request) {
        if(lastModified == 0L) {
          //第一次或者逻辑有变化的时候，应该重新返回内容最新修
改的时间戳
            lastModified = System.currentTimeMillis();
        }
        return lastModified;
    }
}
```

HelloWorldLastModifiedCacheController只需要实现LastModified接口的getLastModified方法，保证当内容发生改变时返回最新的修改时间即可。

Spring 判断是否过期，通过判断请求的“If-Modified-Since”是否大于等于当前的 getLast Modified方法的时间戳，如果是，则认为没有修改。上面的controller与普通的controller并无太大差别，声明如下：

```
<bean name="/helloLastModified"
class="com.test.controller.HelloWorldLastModifiedCache Controller"/
```

11.4.6 HandlerInterceptor的处理

Servlet API定义的 servlet过滤器可以在 servlet处理每个Web请求的前后分别对它进行前置处理和后置处理。此外，有些时候，你可能只想处理由某些SpringMVC处理程序处理的Web请求，并在这些处理程序返回的模型属性被传递到视图之前，对它们进行一些操作。

SpringMVC 允许你通过处理拦截 Web请求，进行前置处理和后置处理。处理拦截是在Spring的Web应用程序上下文中配置的，因此它们

可以利用各种容器特性，并引用容器中声明的任何bean。处理拦截是针对特殊的处理程序映射进行注册的，因此它只拦截通过这些处理程序映射的请求。每个处理拦截都必须实现HandlerInterceptor接口，它包含三个需要你实现的回调方法：preHandle()、postHandle()和afterCompletion()。第一个和第二个方法分别是在处理程序处理请求之前和之后被调用的。第二个方法还允许访问返回的ModelAndView对象，因此可以在它里面操作模型属性。最后一个方法是在所有请求处理完成之后被调用的（如视图呈现之后），以下是HandlerInterceptor的简单实现：

```
public class MyTestInterceptor implements HandlerInterceptor{
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response,Object handler)throws Exception{{
        long startTime = System.currentTimeMillis();
       request.setAttribute("startTime",startTime);
        return true;
    }
    public void postHandle(HttpServletRequest
request,HttpServletResponse response,
        Object handler,ModelAndView modelAndView)throws
Exception{
        long startTime = (Long)request.getAttribute("startTime");
       request.removeAttribute("startTime");
        long endTime = System.currentTimeMillis();
       modelAndView.addObject("handlingTime",endTime-startTime);
    }
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response,Object handler,Exception ex)throws
```

Exception{
    }
}

在这个拦截器的 preHandler()方法中，你记录了起始时间，并将它保存到请求属性中。这个方法应该返回true，允许DispatcherServlet继续处理请求。否则，DispatcherServlet会认为这个方法已经处理了请求，直接将响应返回给用户。然后，在postHandler()方法中，从请求属性中加载起始时间，并将它与当前时间进行比较。你可以计算总的持续时间，然后把这个时间添加到模型中，传递给视图。最后，afterCompletion()方法无事可做，空着就可以了。

11.4.7 逻辑处理

对于逻辑处理其实是通过适配器中转调用Handler并返回视图的，对应代码：

```
mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());
```

同样，还是以引导示例为基础进行处理逻辑分析，之前分析过，对于普通的 Web 请求，Spring默认使用SimpleControllerHandlerAdapter类进行处理，我们进入SimpleControllerHandlerAdapter类的handle方法如下：

```
public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response,
Object handler)
throws Exception {
    return ((Controller) handler).handleRequest(request, response);
}
```

但是回顾引导示例中的UserController，我们的逻辑是写在

handleRequestInternal函数中而不是handleRequest函数，所以我们还需要进一步分析这期间所包含的处理流程。

```
public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response)
throws Exception {
// Delegate to WebContentGenerator for checking and preparing.
checkAndPrepare(request, response, this instanceof LastModified);
//如果需要session内的同步执行
    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                //调用用户的逻辑
                 return handleRequestInternal(request, response);
            }
        }
    }
    //调用用户逻辑
    return handleRequestInternal(request, response);
}
```

11.4.8 异常视图的处理

有时候系统运行过程中出现异常，而我们并不希望就此中断对用户的服务，而是至少告知客户当前系统在处理逻辑的过程中出现了异常，甚至告知他们因为什么原因导致的。Spring中的异常处理机制会帮我们完成这个工作。其实，这里 Spring 主要的工作就是将逻辑引导至

HandlerExceptionResolver类的resolveException方法，而HandlerExceptionResolver的使用，我们在讲解WebApplicationContext的初始化的时候已经介绍过了。

```
    protected ModelAndView processHandlerException(HttpServletRequest
request, HttpServletResponse
    response,
    Object handler, Exception ex) throws Exception {
        // Check registered HandlerExceptionResolvers...
        ModelAndView exMv = null;
        for (HandlerExceptionResolver handlerExceptionResolver :
this.handlerException

Resolvers) {

            exMv =handlerExceptionResolver.resolveException(request,
response, handler, ex);
            if (exMv != null) {
                break;
            }
        }
        if (exMv != null) {
            if (exMv.isEmpty()) {
                return null;
            }
            // We might still need view name translation for a plain error
model...
            if (!exMv.hasView()) {
```

```
            exMv.setViewName(getDefaultViewName(request));
        }
        if (logger.isDebugEnabled()) {
            logger.debug("Handler execution resulted in exception -
forwarding to
            resolved error view: " + exMv, ex);
        }
        WebUtils.exposeErrorRequestAttributes(request, ex,
getServletName());
        return exMv;
    }
    throw ex;
}
```

11.4.9 根据视图跳转页面

无论是一个系统还是一个站点，最重要的工作都是与用户进行交互，用户操作系统后无论下发的命令成功与否都需要给用户一个反馈，以便于用户进行下一步的判断。所以，在逻辑处理的最后一定会涉及一个页面跳转的问题。

```
protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse
response) throws Exception {
    // Determine locale for request and apply it to the response.
    Locale locale = this.localeResolver.resolveLocale(request);
    response.setLocale(locale);
    View view;
    if (mv.isReference()) {
```

```
            // We need to resolve the view name.
            view =resolveViewName(mv.getViewName(),
mv.getModelInternal(), locale, request);
            if (view == null) {
               throw new ServletException(
               "Could not resolve view with name '" + mv.getViewName() + "'
               in servlet with name '" +
               getServletName() + "'");
            }
         }
         else {
            // No need to lookup: the ModelAndView object contains the
actual View object.
            view = mv.getView();
            if (view == null) {
               throw new ServletException("ModelAndView [" + mv + "]
neither contains
               a view name nor a " +
               "View object in servlet with name '" + getServletName() + "'");
            }
         }
         // Delegate to the View object for rendering.
         if (logger.isDebugEnabled()) {
            logger.debug("Rendering view [" + view + "] in DispatcherServlet
with name
            '" + getServletName() + "'");
         }
```

```
    view.render(mv.getModelInternal(), request, response);
}
```

1．解析视图名称

在上文中我们提到DispatcherServlet会根据ModelAndView选择合适的视图来进行渲染，而这一功能就是在resolveViewName函数中完成的。

```
protected View resolveViewName(String viewName, Map<String, Object> model, Locale locale,
HttpServletRequest request) throws Exception {
    for (ViewResolver viewResolver : this.viewResolvers) {
        View view = viewResolver.resolveViewName(viewName, locale);
        if (view != null) {
            return view;
        }
    }
    return null;
}
```

我们以org.Springframework.web.servlet.view.InternalResourceViewResolver 为例来分析ViewResolver逻辑的解析过程，其中resolveViewName函数的实现是在其父类AbstractCaching ViewResolver中完成的。

```
public View resolveViewName(String viewName, Locale locale) throws Exception {
    if (!isCache()) {
        //不存在缓存的情况下直接创建视图
        return createView(viewName, locale);
    }
```

```
else {
  //直接从缓存中提取
   Object cacheKey = getCacheKey(viewName, locale);
   synchronized (this.viewCache) {
      View view = this.viewCache.get(cacheKey);
      if (view == null && (!this.cacheUnresolved || !this.viewCache.
      containsKey(cacheKey))) {
         // Ask the subclass to create the View object.
         view = createView(viewName, locale);
         if (view != null || this.cacheUnresolved) {
            this.viewCache.put(cacheKey, view);
            if (logger.isTraceEnabled()) {
               logger.trace("Cached view [" + cacheKey + "]");
            }
         }
      }
      return view;
   }
}
}
```

在父类UrlBasedViewResolver中重写了createView函数。

```
protected View createView(String viewName, Locale locale) throws
Exception {
   //如果当前解析器不支持当前解析器如viewName为空等情况
   if (!canHandle(viewName, locale)) {
      return null;
   }
```

```
//处理前缀为redirect:xx的情况
 if (viewName.startsWith(REDIRECT_URL_PREFIX)) {
    String redirectUrl =
viewName.substring(REDIRECT_URL_PREFIX.length());
    RedirectView view = new RedirectView(redirectUrl,
isRedirectContext
    Relative(), isRedirectHttp10Compatible());
    return applyLifecycleMethods(viewName, view);
 }
//处理前缀为forward：xx的情况
 if (viewName.startsWith(FORWARD_URL_PREFIX)) {
    String forwardUrl =
viewName.substring(FORWARD_URL_PREFIX.length());
    return new InternalResourceView(forwardUrl);
 }
 // Else fall back to superclass implementation: calling loadView.
 return super.createView(viewName, locale);
}
protected View createView(String viewName, Locale locale) throws
Exception {
 return loadView(viewName, locale);
}
protected View loadView(String viewName, Locale locale) throws
Exception {
 AbstractUrlBasedView view = buildView(viewName);
 View result = applyLifecycleMethods(viewName, view);
 return (view.checkResource(locale) ? result : null);
```

```
}
protected AbstractUrlBasedView buildView(String viewName) throws
Exception {
    AbstractUrlBasedView view = (AbstractUrlBasedView)
BeanUtils.instantiateClass
   (getViewClass());
//添加前缀以及后缀
    view.setUrl(getPrefix() + viewName + getSuffix());
    String contentType = getContentType();
    if (contentType != null) {
      //设置ContentType
      view.setContentType(contentType);
    }
   view.setRequestContextAttribute(getRequestContextAttribute());
   view.setAttributesMap(getAttributesMap());
    if (this.exposePathVariables != null) {
      view.setExposePathVariables(exposePathVariables);
    }
    return view;
}
```

通读以上代码，我们发现对于InternalResourceViewResolver所提供的解析功能主要考虑到了几个方面的处理。

基于效率的考虑，提供了缓存的支持。

提供了对redirect:xx和forward:xx前缀的支持。

添加了前缀及后缀，并向View中加入了必需的属性设置。

2．页面跳转

当通过viewName解析到对应的View后，就可以进一步地处理跳转

逻辑了。

```
public void render(Map<String, ?> model, HttpServletRequest request,
HttpServletResponse
response) throws Exception {
    if (logger.isTraceEnabled()) {
        logger.trace("Rendering view with name '" + this.beanName + "'
with model " + model +
        " and static attributes " + this.staticAttributes);
    }
    Map<String, Object> mergedModel =
createMergedOutputModel(model, request,
   response);
    prepareResponse(request, response);
   renderMergedOutputModel(mergedModel, request, response);
}
```

在引导示例中，我们了解到对于ModelView的使用，可以将一些属性直接放入其中，然后在页面上直接通过JSTL语法或者原始的request获取。这是一个很方便也很神奇的功能，但是实现却并不复杂，无非是把我们将要用到的属性放入 request 中，以便在其他地方可以直接调用，而解析这些属性的工作就是在createMergedOutputModel函数中完成的。

```
protected Map<String, Object> createMergedOutputModel(Map<String,
?> model, HttpServletRequest
request,
    HttpServletResponse response) {
@SuppressWarnings("unchecked")
Map<String, Object> pathVars = this.exposePathVariables ?
    (Map<String, Object>)
```

```
request.getAttribute(View.PATH_VARIABLES) : null;
        // Consolidate static and dynamic model attributes.
        int size = this.staticAttributes.size();
        size += (model != null) ? model.size() : 0;
        size += (pathVars != null) ? pathVars.size() : 0;
        Map<String, Object> mergedModel = new HashMap<String, Object>
(size);
        mergedModel.putAll(this.staticAttributes);
        if (pathVars != null) {
            mergedModel.putAll(pathVars);
        }
        if (model != null) {
            mergedModel.putAll(model);
        }
        // Expose RequestContext?
            if (this.requestContextAttribute != null) {
                mergedModel.put(this.requestContextAttribute,
createRequestContext(request,
                response, mergedModel));
            }
            return mergedModel;
        }
        //处理页面跳转
        protected void renderMergedOutputModel(
            Map<String, Object> model, HttpServletRequest request,
HttpServletResponse
            response) throws Exception {
```

```
            // Determine which request handle to expose to the
RequestDispatcher.
            HttpServletRequest requestToExpose =
getRequestToExpose(request);
        //将model中的数据以属性的方式设置到request中
            exposeModelAsRequestAttributes(model, requestToExpose);
            // Expose helpers as request attributes, if any.
           exposeHelpers(requestToExpose);
            // Determine the path for the request dispatcher.
            String dispatcherPath = prepareForRendering(requestToExpose,
response);
            // Obtain a RequestDispatcher for the target resource (typically a
JSP).
            RequestDispatcher rd = getRequestDispatcher(requestToExpose,
dispatcherPath);
            if (rd == null) {
                throw new ServletException("Could not get RequestDispatcher for
[" + getUrl() +
                "]: Check that the corresponding file exists within your web
                application archive!");
           }
            // If already included or response already committed, perform
include, else forward.
            if (useInclude(requestToExpose, response)) {
               response.setContentType(getContentType());
                if (logger.isDebugEnabled()) {
                    logger.debug("Including resource [" + getUrl() + "] in Internal
```

```
            ResourceView '" + getBeanName() + "'");
        }
        rd.include(requestToExpose, response);
    }
    else {
        // Note: The forwarded resource is supposed to determine the
content type itself.
        exposeForwardRequestAttributes(requestToExpose);
        if (logger.isDebugEnabled()) {
            logger.debug("Forwarding to resource [" + getUrl() + "] in
            InternalResourceView '" + getBeanName() + "'");
        }
        rd.forward(requestToExpose, response);
    }
}
```

第12章　远程服务

Java远程方法调用，即Java RMI（Java Remote Method Invocation），是Java编程语言里一种用于实现远程过程调用的应用程序编程接口。它使客户机上运行的程序可以调用远程服务器上的对象。远程方法调用特性使Java编程人员能够在网络环境中分布操作。RMI全部的宗旨就是尽可能地简化远程接口对象的使用。

Java RMI极大地依赖于接口。在需要创建一个远程对象时，程序员通过传递一个接口来隐藏底层的实现细节。客户端得到的远程对象句柄正好与本地的根代码连接，由后者负责透过网络通信。这样一来，程序员只需关心如何通过自己的接口句柄发送消息。

12.1 RMI

在Spring中，同样提供了对RMI的支持，使得在Spring下的RMI开发变得更方便，同样，我们还是通过示例来快速体验RMI所提供的功能。

12.1.1 使用示例

以下提供了Spring整合RMI的使用示例。

（1）建立RMI对外接口。

```
public interface HelloRMIService {
    public int getAdd(int a, int b);
}
```

（2）建立接口实现类。

```
public class HelloRMIServiceImpl implements HelloRMIService {
    public int getAdd(int a, int b) {
        return a + b;
    }
}
```

（3）建立服务端配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
http://www.Springframework.org/schema/beans
http://www.Springframework.org/schema/beans/Spring-beans-3.0.xsd
">
<!--服务端-->
    <bean id="helloRMIServiceImpl"
```

```
class="test.remote.HelloRMIServiceImpl" />
        <!--将类为一个RMI服务 -->
        <bean id="myRMI"
class="org.Springframework.remoting.RMI.RMIServiceExporter">
            <!--服务类 -->
            <property name="service" ref="helloRMIServiceImpl" />
            <!--服务名 -->
            <property name="serviceName" value="helloRMI" />
            <!--服务接口 -->
            <property name="serviceInterface"
value="test.remote.HelloRMIService" />
            <!--服务端口 -->
            <property name="registryPort" value="9999" />
            <!-- 其他属性自己查看
org.Springframework.remoting.RMI.RMIServiceExporter 的类,就
    知道支持的属性了-->
        </bean>
    </beans>
```

（4）建立服务端测试。

```
    public class ServerTest {
        public static void main(String[] args) {
            new
ClassPathXmlApplicationContext("test/remote/RMIServer.xml");
        }
    }
```

到这里，建立 RMI 服务端的步骤已经结束了，服务端发布了一个两数相加的对外接口供其他服务器调用。启动服务端测试类，其他机器

或端口便可以通过RMI来连接到本机了。

（5）完成了服务端的配置后，还需要在测试端建立测试环境以及测试代码。首先建立测试端配置文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
http://www.Springframework.org/schema/beans
http://www.Springframework.org/schema/beans/Spring-beans-3.0.xsd
">
   <!--客户端-->
    <bean id="myClient"
class="org.Springframework.remoting.RMI.RMIProxyFactoryBean">
       <property name="serviceUrl"
value="RMI://127.0.0.1:9999/helloRMI"/>
       <property name="serviceInterface"
value="test.remote.HelloRMIService"/>
   </bean>
</beans>
```

（6）编写测试代码。

```
public class ClientTest {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext ("test/remote/
       RMIClient.xml");
        HelloRMIService hms = context.getBean("myClient",
HelloRMIService.class);
```

```
            System.out.println(hms.getAdd(1, 2));
        }
    }
```

通过以上的步骤，实现了测试端的代码调用。你会看到测试端通过RMI进行了远程连接，连接到了服务端，并使用对应的实现类HelloRMIServiceImpl中提供的方法getAdd来计算参数并返回结果，你会看到控制台输出了3。当然以上的测试用例是使用同一台机器不同的端口来模拟不同机器的RMI连接。在企业应用中一般都是使用不同的机器来进行RMI服务的发布与访问，你需要将接口打包，并放置在服务端的工程中。

这是一个简单的方法展示，但是却很好地展示了Spring中使用RMI的流程以及步骤，如果抛弃Spring而使用原始的RMI发布与连接，则会是一件很麻烦的事情，有兴趣的读者可以查阅相关的资料。在Spring中使用RMI非常简单，Spring帮助我们做了大量的工作，这些工作都包括什么呢？接下来我们一起深入分析Spring中对RMI功能的实现原理。

12.1.2 服务端实现

首先我们从服务端的发布功能开始着手，同样，Spring中的核心还是配置文件，这是所有功能的基础。在服务端的配置文件中我们可以看到，定义了两个 bean，其中一个是对接口实现类的发布，而另一个则是对RMI服务的发布，使用org.Springframework.remoting.RMI.RMIServiceExporter类进行封装，其中包括了服务类、服务名、服务接口、服务端口等若干属性，因此我们可以断定， org.Springframework.remoting.RMI.RMIServiceExporter 类应该是发布 RMI 的关键类。我们可以从此类入手进行分析。

根据前面展示的示例，启动Spring中的RMI服务并没有多余的操作，仅仅是开启Spring的环境：new

ClassPathXmlApplicationContext("test/remote/RMIServer.xml")，仅此一句。于是，我们分析很可能是RMIServiceExportern在初始化的时候做了某些操作完成了端口的发布功能，那么这些操作的入口是在这个类的哪个方法里面呢？

进入这个类，首先分析这个类的层次结构，如图12-1所示。

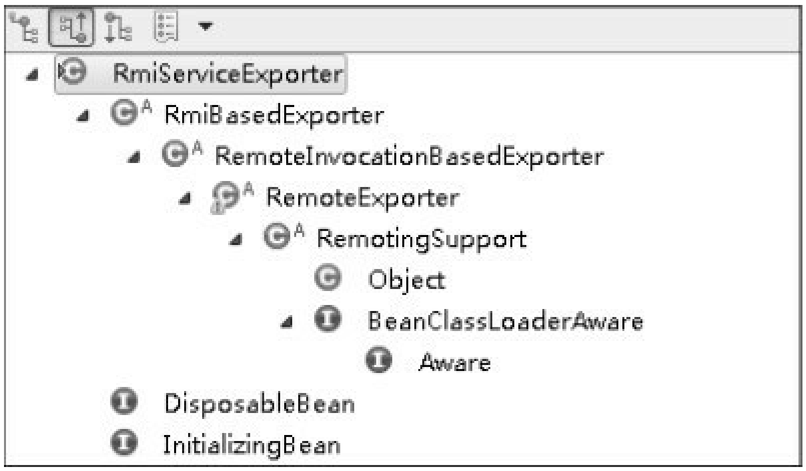


图12-1 RMIServiceExporter类层次结构图

根据Eclipse提供的功能，我们查看到了RMIServiceExporter的层次结构图，那么从这个层次图中我们能得到什么信息呢？

RMIServiceExporter 实现了 Spring 中几个比较敏感的接口：BeanClassLoaderAware、DisposableBean、InitializingBean，其中，DisposableBean接口保证在实现该接口的bean销毁时调用其destroy方法，BeanClassLoaderAware接口保证在实现该接口的bean的初始化时调用其setBeanClassLoader方法，而InitializingBean接口则是保证在实现该接口的bean初始化时调用其 afterPropertiesSet 方法，所以我们推断RMIServiceExporter 的初始化函数入口一定在其afterPropertiesSet 或者setBeanClassLoader 方法中。经过查看代码，确认 afterPropertiesSet 为RMIServiceExporter功能的初始化入口。

```
public void afterPropertiesSet() throws RemoteException {
    prepare();
}
public void prepare() throws RemoteException {
    //检查验证service
    checkService();
     if (this.serviceName == null) {
        throw new IllegalArgumentException("Property 'serviceName' is 
required");
    }
    //如果用户在配置文件中配置了clientSocketFactory或者
serverSocketFactory的处理
    /*
        * 如果配置中的clientSocketFactory同时又实现了
RMIServerSocketFactory接口那么会忽略
        * 配置中的serverSocketFactory而使用clientSocketFactory代替
        */
     if (this.clientSocketFactory instanceof RMIServerSocketFactory) {
        this.serverSocketFactory = (RMIServerSocketFactory) 
this.clientSocketFactory;
    }
    //clientSocketFactory和serverSocketFactory要么同时出现要么都不
出现
     if ((this.clientSocketFactory != null && this.serverSocketFactory == 
null) ||
     (this.clientSocketFactory == null && this.serverSocketFactory != 
null)) {
```

```
        throw new IllegalArgumentException(
        "Both RMIClientSocketFactory and RMIServerSocketFactory or
none
       required");
    }
    /*
       * 如果配置中的registryClientSocketFactory同时实现了
RMIServerSocketFactory接口那么
       * 会忽略配置中的registryServerSocketFactory而使用
registryClientSocketFactory代替
       */
     if (this.registryClientSocketFactory instanceof
RMIServerSocketFactory) {
        this.registryServerSocketFactory = (RMIServerSocketFactory)
this.registry
       ClientSocketFactory;
    }
    //不允许出现只配置registryServerSocketFactory却没有配置
registryClientSocketFactory的
    情况出现
     if (this.registryClientSocketFactory == null &&
this.registryServerSocket
     Factory != null) {
        throw new IllegalArgumentException(
        "RMIServerSocketFactory without RMIClientSocketFactory for
        registry not supported");
    }
```

```
    this.createdRegistry = false;
    //确定RMI registry
    if (this.registry == null) {
        this.registry = getRegistry(this.registryHost, this.registryPort,
       this.registryClientSocketFactory,
       this.registryServerSocketFactory);
        this.createdRegistry = true;
    }
    //初始化以及缓存导出的Object
    //此时通常情况下是使用RMIInvocationWrapper封装的JDK代理
类，切面为RemoteInvocation
    TraceInterceptor
    this.exportedObject = getObjectToExport();
    if (logger.isInfoEnabled()) {
        logger.info("Binding service '" + this.serviceName + "' to RMI
registry: "
        + this.registry);
    }
    // Export RMI object.
    if (this.clientSocketFactory != null) {
       /*
          * 使用由给定的套接字工厂指定的传送方式导出远程对象，
以便能够接收传入的调用。
          * clientSocketFactory:进行远程对象调用的客户端套接字工厂
          * serverSocketFactory:接收远程调用的服务端套接字工厂
          */
       UnicastRemoteObject.exportObject(
```

```
        this.exportedObject, this.servicePort, this.clientSocketFactory,
       this.serverSocketFactory);
    }
    else {
        //导出remote object,以使它能接收特定端口的调用
        UnicastRemoteObject.exportObject(this.exportedObject,
this.servicePort);
    }
    try {
        if (this.replaceExistingBinding) {
            this.registry.rebind(this.serviceName, this.exportedObject);
        }
        else {
            //绑定服务名称到remote object，外界调用 serviceName的时
候会被 exportedObject
           接收
            this.registry.bind(this.serviceName, this.exportedObject);
        }
    }
    catch (AlreadyBoundException ex) {
       unexportObjectSilently();
        throw new IllegalStateException(
        "Already an RMI object bound for name '" + this.serviceName +
"':
        " + ex.toString());
    }
    catch (RemoteException ex) {
```

```
            unexportObjectSilently();
             throw ex;
        }
    }
```

果然，在afterPropertiesSet函数中将实现委托给了prepare，而在prepare方法中我们找到了RMI服务发布的功能实现，同时，我们也大致清楚了RMI服务发布的流程。

（1）验证service。

此处的service对应的是配置中类型为RMIServiceExporter的service属性，它是实现类，并不是接口。尽管后期会对RMIServiceExporter做一系列的封装，但是，无论怎么封装，最终还是会将逻辑引向至RMIServiceExporter来处理，所以，在发布之前需要进行验证。

（2）处理用户自定义的SocketFactory属性。

在RMIServiceExporter中提供了4个套接字工厂配置，分别是clientSocketFactory、serverSocket Factory和registryClientSocketFactory、registryServerSocketFactory。那么这两对配置又有什么区别或者说分别是应用在什么样的不同场景呢？

registryClientSocketFactory与registryServerSocketFactory用于主机与RMI服务器之间连接的创建，也就是当使用LocateRegistry.createRegistry(registryPort, clientSocketFactory, server SocketFactory)方法创建Registry实例时会在RMI主机使用serverSocketFactory创建套接字等待连接，而服务端与RMI主机通信时会使用clientSocketFactory创建连接套接字。

clientSocketFactory、serverSocketFactory 同样是创建套接字，但是使用的位置不同， clientSocketFactory、serverSocketFactory用于导出远程对象，serverSocketFactory用于在服务端建立套接字等待客户端连接，而clientSocketFactory用于调用端建立套接字发起连接。

（3）根据配置参数获取Registry。

（4）构造对外发布的实例。

构建对外发布的实例，当外界通过注册的服务名调用响应的方法时，RMI服务会将请求引入此类来处理。

（5）发布实例。

在发布RMI服务的流程中，有几个步骤可能是我们比较关心的。

1．获取registry

对RMI稍有了解就会知道，由于底层的封装，获取Registry实例是非常简单的，只需要使用一个函数LocateRegistry.createRegistry(...)创建Registry实例就可以了。但是，Spring中并没有这么做，而是考虑得更多，比如 RMI 注册主机与发布的服务并不在一台机器上，那么需要使用LocateRegistry.getRegistry(registryHost, registryPort, clientSocketFactory)去远程获取Registry实例。

```
protected Registry getRegistry(String registryHost, int registryPort,
RMIClientSocketFactory clientSocketFactory, RMIServerSocketFactory
serverSocketFactory)
throws RemoteException {
    if (registryHost != null) {
       //远程连接测试
        if (logger.isInfoEnabled()) {
           logger.info("Looking for RMI registry at port '" + registryPort +
"'
           of host [" + registryHost + "]");
       }
       //如果registryHost不为空则尝试获取对应主机的Registry
        Registry reg = LocateRegistry.getRegistry(registryHost,
registryPort,
```

clientSocketFactory);
testRegistry(reg);
return reg;
}else {
//获取本机的Registry
return getRegistry(registryPort, clientSocketFactory,
serverSocketFactory);
}
}

如果并不是从另外的服务器上获取Registry连接，那么就需要在本地创建RMI的Registry实例了。当然，这里有一个关键的参数alwaysCreateRegistry，如果此参数配置为true，那么在获取Registry实例时会首先测试是否已经建立了对指定端口的连接，如果已经建立则复用已经创建的实例，否则重新创建。

当然，之前也提到过，创建Registry实例时可以使用自定义的连接工厂，而之前的判断也保证了clientSocketFactory与serverSocketFactory要么同时出现，要么同时不出现，所以这里只对clientSocketFactory是否为空进行了判断。

protected Registry getRegistry(
int registryPort, RMIClientSocketFactory clientSocketFactory,
RMIServerSocketFactory
serverSocketFactory)
throws RemoteException {
if (clientSocketFactory != null) {
if (this.alwaysCreateRegistry) {
logger.info("Creating new RMI registry");
//使用clientSocketFactory创建Registry

```
                serverSocketFactory);
                return LocateRegistry.createRegistry(registryPort,
clientSocketFactory,
            }
            if (logger.isInfoEnabled()) {
                logger.info("Looking for RMI registry at port '" + registryPort +
"',
                using custom socket factory");
            }
            synchronized (LocateRegistry.class) {
                try {
                    //复用测试
                    Registry reg = LocateRegistry.getRegistry(null, registryPort,
                    clientSocketFactory);
                    testRegistry(reg);
                    return reg;
                }
                catch (RemoteException ex) {
                    logger.debug("RMI registry access threw exception", ex);
                    logger.info("Could not detect RMI registry - creating new
one");
                    return LocateRegistry.createRegistry(registryPort,
clientSocketFactory,
                    serverSocketFactory);
                }
            }
        }else {
```

```
            return getRegistry(registryPort);
        }
    }
```

如果创建 Registry 实例时不需要使用自定义的套接字工厂，那么就可以直接使用LocateRegistry.createRegistry(...)方法来创建了，当然复用的检测还是必要的。

```
    protected Registry getRegistry(int registryPort) throws
RemoteException {
        if (this.alwaysCreateRegistry) {
            logger.info("Creating new RMI registry");
            return LocateRegistry.createRegistry(registryPort);
        }
        if (logger.isInfoEnabled()) {
            logger.info("Looking for RMI registry at port '" + registryPort +
"'");
        }
        synchronized (LocateRegistry.class) {
            try {
              //查看对应当前registryPort的Registry是否已经创建，如果创
建直接使用
               Registry reg = LocateRegistry.getRegistry(registryPort);
              //测试是否可用，如果不可用则抛出异常
              testRegistry(reg);
               return reg;
            }
            catch (RemoteException ex) {
               logger.debug("RMI registry access threw exception", ex);
```

```
            logger.info("Could not detect RMI registry - creating new one");
           //根据端口创建Registry
            return LocateRegistry.createRegistry(registryPort);
        }
    }
}
```

2. 初始化将要导出的实体对象

之前有提到过，当请求某个RMI服务的时候，RMI 会根据注册的服务名称，将请求引导至远程对象处理类中，这个处理类便是使用getObjectToExport()进行创建。

```
protected Remote getObjectToExport() {
    //如果配置的service属性对应的类实现了Remote接口且没有配置serviceInterface属性
     if (getService() instanceof Remote &&
     (getServiceInterface() == null || Remote.class.isAssignableFrom
     (getServiceInterface()))) {
         return (Remote) getService();
     }
     else {
         if (logger.isDebugEnabled()) {
             logger.debug("RMI service [" + getService() + "] is an RMI invoker");
         }
        //对service进行封装
         return new RMIInvocationWrapper(getProxyForService(), this);
     }
}
```

请求处理类的初始化主要处理规则为：如果配置的service属性对应的类实现了Remote接口且没有配置 serviceInterface 属性，那么直接使用 service 作为处理类；否则，使用RMIInvocationWrapper对service的代理类和当前类也就是RMIServiceExporter进行封装。

经过这样的封装，客户端与服务端便可以达成一致协议，当客户端检测到是RMIInvocation Wrapper类型stub的时候便会直接调用其invoke方法，使得调用端与服务端很好地连接在了一起。而 RMIInvocationWrapper 封装了用于处理请求的代理类，在 invoke 中便会使用代理类进行进一步处理。

之前的逻辑已经非常清楚了，当请求RMI服务时会由注册表Registry实例将请求转向之前注册的处理类去处理，也就是之前封装的RMIInvocationWrapper，然后由RMIInvocationWrapper中的invoke方法进行处理，那么为什么不是在invoke方法中直接使用service，而是通过代理再次将service封装呢？

这其中的一个关键点是，在创建代理时添加了一个增强拦截器RemoteInvocationTraceInterceptor，目的是为了对方法调用进行打印跟踪，但是如果直接在 invoke 方法中硬编码这些日志，会使代码看起来很不优雅，而且耦合度很高，使用代理的方式就会解决这样的问题，而且会有很高的可扩展性。

```
protected Object getProxyForService() {
    //验证service
    checkService();
    //验证serviceInterface
    checkServiceInterface();
    //使用JDK的方式创建代理
     ProxyFactory proxyFactory = new ProxyFactory();
    //添加代理接口
```

```
proxyFactory.addInterface(getServiceInterface());
 if (this.registerTraceInterceptor != null ?
 this.registerTraceInterceptor.booleanValue() : this.interceptors ==
null) {
   //加入代理的横切面RemoteInvocationTraceInterceptor并记录
Exporter名称
   proxyFactory.addAdvice(new
   RemoteInvocationTraceInterceptor(getExporterName()));
}
 if (this.interceptors != null) {
   AdvisorAdapterRegistry adapterRegistry =
GlobalAdvisorAdapterRegistry.
   getInstance();
   for (int i = 0; i < this.interceptors.length; i++) {

proxyFactory.addAdvisor(adapterRegistry.wrap(this.interceptors[i]));
   }
}
//设置要代理的目标类
proxyFactory.setTarget(getService());
proxyFactory.setOpaque(true);
//创建代理
 return proxyFactory.getProxy(getBeanClassLoader());
}
```

3．RMI服务激活调用

之前反复提到过，由于在之前 bean 初始化的时候做了服务名称绑定 this.registry.bind (this.serviceName, this.exportedObject)，其中的

exportedObject其实是被RMIInvocationWrapper进行过封装的，也就是说当其他服务器调用serviceName的RMI服务时，Java会为我们封装其内部操作，而直接会将代码转向RMIInvocationWrapper的invoke方法中。

```
    public Object invoke(RemoteInvocation invocation)
    throws RemoteException, NoSuchMethodException,
IllegalAccessException,
    InvocationTargetException {
        return this.RMIExporter.invoke(invocation, this.wrappedObject);
    }
```

而此时this.RMIExporter为之前初始化的RMIServiceExporter，invocation为包含着需要激活的方法参数，而wrappedObject则是之前封装的代理类。

```
    protected Object invoke(RemoteInvocation invocation, Object
targetObject)
    throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException {
        return super.invoke(invocation, targetObject);
    }
    protected Object invoke(RemoteInvocation invocation, Object
targetObject)
    throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException {
        if (logger.isTraceEnabled()) {
            logger.trace("Executing " + invocation);
        }
        try {
            return getRemoteInvocationExecutor().invoke(invocation,
```

```
targetObject);
            }
            catch (NoSuchMethodException ex) {
                if (logger.isDebugEnabled()) {
                    logger.warn("Could not find target method for " + invocation,
ex);
                }
                throw ex;
            }
            catch (IllegalAccessException ex) {
                if (logger.isDebugEnabled()) {
                    logger.warn("Could not access target method for " + invocation,
ex);
                }
                throw ex;
            }
            catch (InvocationTargetException ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug("Target method failed for " + invocation,
ex.getTargetException());
                }
                throw ex;
            }
        }
        public Object invoke(RemoteInvocation invocation, Object
targetObject)
        throws NoSuchMethodException, IllegalAccessException,
```

```
InvocationTargetException{
        Assert.notNull(invocation, "RemoteInvocation must not be null");
        Assert.notNull(targetObject, "Target object must not be null");
    //通过反射方式激活方法
        return invocation.invoke(targetObject);
    }
    public Object invoke(Object targetObject)
    throws NoSuchMethodException, IllegalAccessException,
InvocationTarget
    Exception {
      //根据方法名称获取代理中对应的方法
        Method method =
targetObject.getClass().getMethod(this.methodName, this.
      parameterTypes);
      //执行代理中的方法
        return method.invoke(targetObject, this.arguments);
    }
```

12.1.3 客户端实现

根据客户端配置文件，锁定入口类为RMIProxyFactoryBean，同样根据类的层次结构查找入口函数，如图12-2所示。

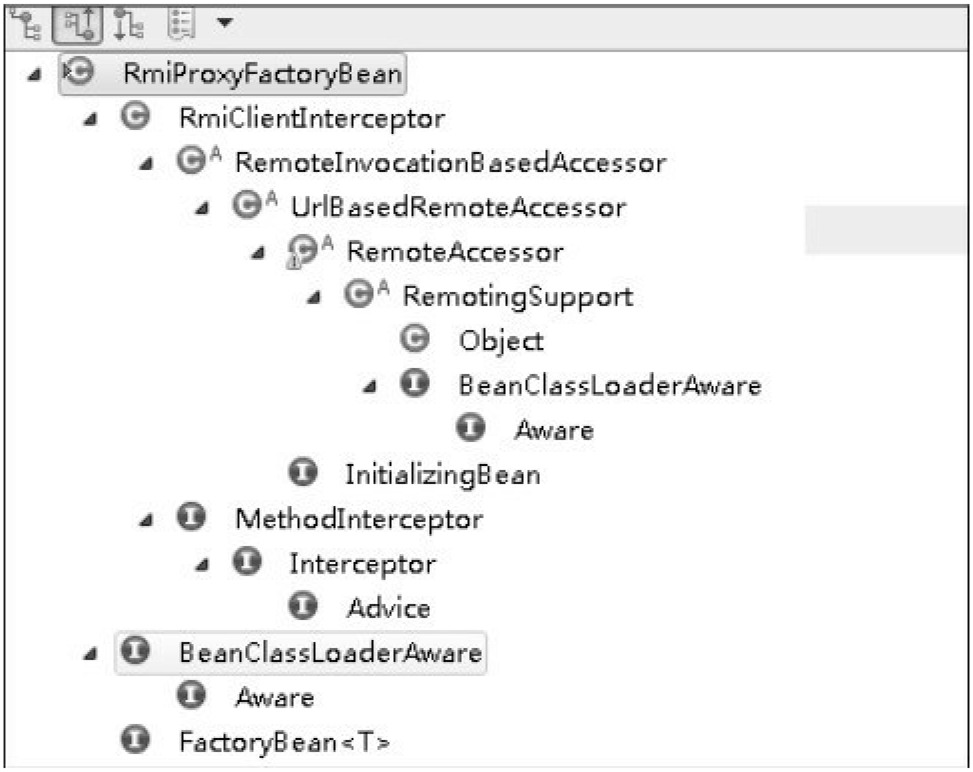


图12-2 RMIProxyFactoryBean类的层次结构图

根据层次关系以及之前的分析，我们提取出该类实现的比较重要的接口 InitializingBean、BeanClassLoaderAware以及MethodInterceptor。

其中实现了InitializingBean，则Spring会确保在此初始化bean时调用afterPropertiesSet进行逻辑的初始化。

```
public void afterPropertiesSet() {
    super.afterPropertiesSet();
     if (getServiceInterface() == null) {
        throw new IllegalArgumentException("Property 'serviceInterface' is
       required");
    }
    //根据设置的接口创建代理，并使用当前类this作为增强器
```

```
    this.serviceProxy = new ProxyFactory(getServiceInterface(), this).getProxy
   (getBeanClassLoader());
}
```

同时，RMIProxyFactoryBean又实现了FactoryBean接口，那么当获取bean时并不是直接获取bean，而是获取该bean的getObject方法。

```
public Object getObject() {
    return this.serviceProxy;
}
```

这样，我们似乎已近形成了一个大致的轮廓，当获取该bean时，首先通过afterPropertiesSet创建代理类，并使用当前类作为增强方法，而在调用该 bean 时其实返回的是代理类，既然调用的是代理类，那么又会使用当前 bean 作为增强器进行增强，也就是说会调用RMIProxyFactoryBean的父类RMIClientInterceptor的invoke方法。

我们先从afterPropertiesSet中的super.afterPropertiesSet()方法开始分析。

```
public void afterPropertiesSet() {
   super.afterPropertiesSet();
   prepare();
}
```

继续追踪代码，发现父类的父类，也就是 UrlBasedRemoteAccessor中的 afterPropertiesSet方法只完成了对serviceUrl属性的验证。

```
public void afterPropertiesSet() {
    if (getServiceUrl() == null) {
        throw new IllegalArgumentException("Property 'serviceUrl' is required");
    }
```

}

所以推断所有的客户端都应该在prepare方法中实现，继续查看prepare()。

1．通过代理拦截并获取stub

在父类的afterPropertiesSet方法中完成了对serviceUrl的验证，那么prepare函数又完成了什么功能呢？

```
public void prepare() throws RemoteLookupFailureException {
    // Cache RMI stub on initialization?
   //如果配置了lookupStubOnStartup属性便会在启动时寻找stub
    if (this.lookupStubOnStartup) {
        Remote remoteObj = lookupStub();
        if (logger.isDebugEnabled()) {
            if (remoteObj instanceof RMIInvocationHandler) {
                logger.debug("RMI stub [" + getServiceUrl() + "] is an RMI
               invoker");
            }
            else if (getServiceInterface() != null) {
                boolean isImpl = getServiceInterface().isInstance(remoteObj);
                logger.debug("Using service interface [" +
getServiceInterface().
                getName() +
                "] for RMI stub [" + getServiceUrl() + "] - " +
                (!isImpl ? "not " : "") + "directly implemented");
            }
        }
        if (this.cacheStub) {
           //将获取的stub缓存
```

```
            this.cachedStub = remoteObj;
        }
    }
}
```

从上面的代码中，我们了解到了一个很重要的属性lookupStubOnStartup，如果将此属性设置为true，那么获取stub的工作就会在系统启动时被执行并缓存，从而提高使用时候的响应时间。

获取stub是RMI应用中的关键步骤，当然你可以使用两种方式进行。

（1）使用自定义的套接字工厂。如果使用这种方式，你需要在构造 Registry 实例时将自定义套接字工厂传入，并使用Registry中提供的lookup方法来获取对应的stub。

（2）直接使用RMI提供的标准方法：Naming.lookup(getServiceUrl())。

```
protected Remote lookupStub() throws RemoteLookupFailureException
{
    try {
        Remote stub = null;
        if (this.registryClientSocketFactory != null) {
            URL url = new URL(null, getServiceUrl(), new
DummyURLStreamHandler());
            String protocol = url.getProtocol();
            //验证传输协议
            if (protocol != null && !"RMI".equals(protocol)) {
                throw new MalformedURLException("Invalid URL scheme
'" + protocol + "'");
            }
```

```
                //主机
                 String host = url.getHost();
                //端口
                 int port = url.getPort();
                //服务名
                 String name = url.getPath();
                 if (name != null && name.startsWith("/")) {
                     name = name.substring(1);
                }
                 Registry registry = LocateRegistry.getRegistry(host, port,
this.registry
                 Client SocketFactory);
                 stub = registry.lookup(name);
             }
             else {
                 // Can proceed with standard RMI lookup API...
                 stub = Naming.lookup(getServiceUrl());
             }
             if (logger.isDebugEnabled()) {
                 logger.debug("Located RMI stub with URL [" + getServiceUrl()
+ "]");
             }
             return stub;
         }
         catch (MalformedURLException ex) {
             throw new RemoteLookupFailureException("Service URL [" +
getServiceUrl() +
```

```
            "] is invalid", ex);
        }
        catch (NotBoundException ex) {
            throw new RemoteLookupFailureException(
            "Could not find RMI service [" + getServiceUrl() + "] in RMI
            registry", ex);
        }
        catch (RemoteException ex) {
            throw new RemoteLookupFailureException("Lookup of RMI stub
failed", ex);
        }
    }
```

为了使用registryClientSocketFactory，代码量比使用RMI标准获取stub方法多出了很多，那么registryClientSocketFactory到底是做什么用的呢？

与之前服务端的套接字工厂类似，这里的registryClientSocketFactory用来连接RMI服务器，用户通过实现RMIClientSocketFactory接口来控制用于连接的socket的各种参数。

2．增强器进行远程连接

之前分析了类型为RMIProxyFactoryBean的bean的初始化中完成的逻辑操作。在初始化时，创建了代理并将本身作为增强器加入了代理中（RMIProxyFactoryBean间接实现了MethodInterceptor）。那么这样一来，当在客户端调用代理的接口中的某个方法时，就会首先执行RMIProxyFactoryBean中的invoke方法进行增强。

```
    public Object invoke(MethodInvocation invocation) throws Throwable {
       //获取的服务器中对应的注册的remote对象，通过序列化传输
        Remote stub = getStub();
```

```
        try {
            return doInvoke(invocation, stub);
        }
        catch (RemoteConnectFailureException ex) {
            return handleRemoteConnectFailure(invocation, ex);
        }
        catch (RemoteException ex) {
            if (isConnectFailure(ex)) {
                return handleRemoteConnectFailure(invocation, ex);
            }
            else {
                throw ex;
            }
        }
    }
```

众所周知，当客户端使用接口进行方法调用时是通过RMI获取stub的，然后再通过stub中封装的信息进行服务器的调用，这个stub就是在构建服务器时发布的对象，那么，客户端调用时最关键的一步也是进行stub的获取了。

```
    protected Remote getStub() throws RemoteLookupFailureException {
        if (!this.cacheStub || (this.lookupStubOnStartup &&
!this.refreshStubOnConnect
        Failure)) {
          //如果有缓存直接使用缓存
        return (this.cachedStub != null ? this.cachedStub : lookupStub());
        }
        else {
```

```
            synchronized (this.stubMonitor) {
                if (this.cachedStub == null) {
               //获取stub
                    this.cachedStub = lookupStub();
                }
                return this.cachedStub;
            }
        }
    }
```

当获取到stub后便可以进行远程方法的调用了。Spring中对于远程方法的调用其实是分两种情况考虑的。

获取的stub是RMIInvocationHandler类型的，从服务端获取的stub是RMIInvocation Handler，就意味着服务端也同样使用了 Spring 去构建，那么自然会使用 Spring 中作的约定，进行客户端调用处理。Spring中的处理方式被委托给了doInvoke方法。

当获取的stub不是RMIInvocationHandler类型，那么服务端构建RMI服务可能是通过普通的方法或者借助于 Spring 外的第三方插件，那么处理方式自然会按照 RMI 中普通的处理方式进行，而这种普通的处理方式无非是反射。因为在 invocation 中包含了所需要调用的方法的各种信息，包括方法名称以及参数等，而调用的实体正是 stub，那么通过反射方法完全可以激活stub中的远程调用。

```
    protected Object doInvoke(MethodInvocation invocation, Remote stub)
throws Throwable {
        //stub从服务器传回且经过Spring的封装
         if (stub instanceof RMIInvocationHandler) {
            try {
                return doInvoke(invocation, (RMIInvocationHandler) stub);
```

```
}
 catch (RemoteException ex) {
    throw RMIClientInterceptorUtils.convertRMIAccessException(
    invocation.getMethod(), ex, isConnectFailure(ex),
getServiceUrl());
}
 catch (InvocationTargetException ex) {
    Throwable exToThrow = ex.getTargetException();

RemoteInvocationUtils.fillInClientStackTraceIfPossible(exToThrow);
    throw exToThrow;
}
 catch (Throwable ex) {
    throw new RemoteInvocationFailureException("Invocation of
method [" +
    invocation.getMethod() +
    "] failed in RMI service [" + getServiceUrl() + "]", ex);
   }
}
 else {
    try {
//直接使用反射方法继续激活
    return
RMIClientInterceptorUtils.invokeRemoteMethod(invocation, stub);
}
 catch (InvocationTargetException ex) {
    Throwable targetEx = ex.getTargetException();
```

```
                if (targetEx instanceof RemoteException) {
                    RemoteException rex = (RemoteException) targetEx;
                    throw
RMIClientInterceptorUtils.convertRMIAccessException(
                    invocation.getMethod(), rex, isConnectFailure(rex),
                   getServiceUrl());
                }
                else {
                    throw targetEx;
                }
            }
        }
    }
```

之前反复提到了Spring中的客户端处理RMI的方式。其实，在分析服务端发布RMI的方式时，我们已经了解到，Spring将RMI的导出Object封装成了RMIInvocationHandler类型进行发布，那么当客户端获取stub的时候是包含了远程连接信息代理类的RMIInvocationHandler，也就是说当调用RMIInvocationHandler中的方法时会使用RMI中提供的代理进行远程连接，而此时，Spring中要做的就是将代码引向RMIInvocationHandler接口的invoke方法的调用。

```
    protected Object doInvoke(MethodInvocation methodInvocation,
RMIInvocationHandler
    invocationHandler)
        throws RemoteException, NoSuchMethodException,
IllegalAccessException, Invocation
        TargetException {
        if (AopUtils.isToStringMethod(methodInvocation.getMethod())) {
```

```
            return "RMI invoker proxy for service URL [" + getServiceUrl() +
"]";
        }
        //将methodInvocation中的方法名及参数等信息重新封装到
RemoteInvocation，并通过远程代理
        方法直接调用
         return
invocationHandler.invoke(createRemoteInvocation(methodInvocation));
    }
```

12.2 HttpInvoker

Spring开发小组意识到在RMI服务和基于HTTP的服务（如Hessian和Burlap）之间的空白。一方面，RMI使用Java标准的对象序列化，但很难穿越防火墙；另一方面，Hessian/Burlap能很好地穿过防火墙工作，但使用自己私有的一套对象序列化机制。

就这样，Spring的HttpInvoker应运而生。HttpInvoker是一个新的远程调用模型，作为Spring框架的一部分，来执行基于HTTP的远程调用（让防火墙高兴的事），并使用Java的序列化机制（这是让程序员高兴的事）。

我们首先看看HttpInvoker的使用示例。HttpInvoker是基于HTTP的远程调用，同时也是使用Spring中提供的web服务作为基础，所以我们的测试需要首先搭建Web工程。

12.2.1 使用示例

（1）创建对外接口。

```
public interface HttpInvokerTestI {
    public String getTestPo(String desp);
```

```
}
```

（2）创建接口实现类。

```
public class HttpInvokertestImpl implements HttpInvokerTestI {
    @Override
     public String getTestPo(String desp) {
         return "getTestPo " + desp;
    }
}
```

（3）创建服务端配置文件applicationContext-server.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://www.Springframework.org/schema/beans
http://www.Springframework.org/schema/beans/Spring-beans-3.0.xsd
">
     <bean name="httpinvokertest" class="test.HttpInvokertestImpl" />
</beans>
```

（4）在WEB-INF下创建remote-servlet.xml。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.Springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://www.Springframework.org/schema/beans
http://www.Springframework.org/schema/beans/Spring-beans-3.0.xsd
">
     <bean name="/hit"
```

```
class="org.Springframework.remoting.httpinvoker.HttpInvokerServiceExport
                <property name="service" ref="httpinvokertest" />
                <property name="serviceInterface" value="test.HttpInvokerTestI"
/>
            </bean>
        </beans>
```

至此，服务端的 httpInvoker 服务已经搭建完了，启动 Web 工程后就可以使用我们搭建的HttpInvoker服务了。以上代码实现了将远程传入的字符串参数处理加入“getTestPo”前缀的功能。服务端搭建完基于Web服务的HttpInvoker后，客户端还需要使用一定的配置才能进行远程调用。

（5）创建测试端配置client.xml。

```
        <?xml version="1.0" encoding="UTF-8"?>
        <beans xmlns="http://www.Springframework.org/schema/beans"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="
        http://www.Springframework.org/schema/beans
        http://www.Springframework.org/schema/beans/Spring-beans-3.0.xsd
">
             <bean id="remoteService"

class="org.Springframework.remoting.httpinvoker.HttpInvokerProxyFactoryB
                <property name="serviceUrl"
value="http://localhost:8080/httpinvokertest/
                remoting/hit" />
                <property name="serviceInterface" value="test.HttpInvokerTestI"
```

```
/>
        </bean>
    </beans>
```

（6）创建测试类。

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext ("classpath:
        client.xml");
        HttpInvokerTestI httpInvokerTestI = (HttpInvokerTestI)
context.getBean
        ("remoteService");
        System.out.println(httpInvokerTestI.getTestPo("dddd"));
    }
}
```

运行测试类，你会看到打印结果：

getTestPo dddd

dddd是我们传入的参数，而getTestPo则是在服务端添加的字符串。当然，上面的服务搭建与测试过程中都是在一台机器上进行的，如果需要在不同机器上进行测试，还需要读者对服务端的相关接口打成JAR包并加入到客户端的服务器上。

12.2.2 服务端实现

对于Spring中HttpInvoker服务的实现，我们还是首先从服务端进行分析。

根据remote-servlet.xml中的配置，我们分析入口类应该为org.Springframework.remoting.httpinvoker. HttpInvokerServiceExporter，

那么同样，根据这个类分析其入口函数，如图12-3所示。

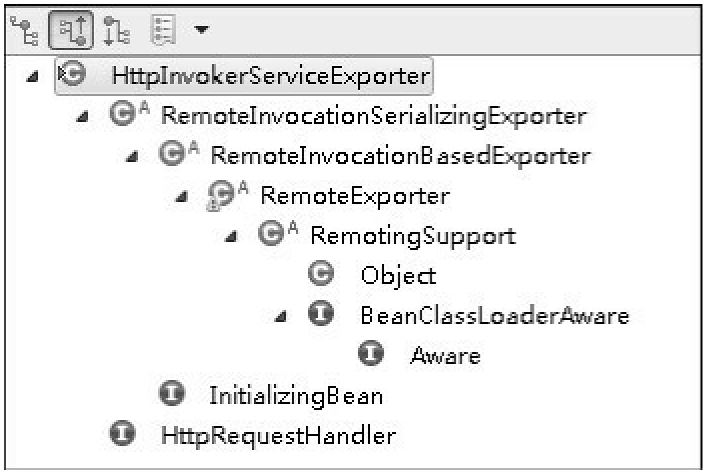


图12-3 HttpInvokerServiceExporter类的层次结构图

通过层次关系我们看到 HttpInvokerService Exporter 类实现了 InitializingBean 接口以及 Http RequestHandler接口。分析RMI服务时我们已经了解到了，当某个bean继承自InitializingBean接口的时候，Spring 会确保这个 bean 在初始化时调用其 afterPropertiesSet 方法，而对于HttpRequestHandler接口，因为我们在配置中已经将此接口配置成Web服务，那么当有相应请求的时候，Spring的Web服务就会将程序引导至HttpRequestHandler的handleRequest方法中。首先，我们从afterPropertiesSet方法开始分析，看看在bean的初始化过程中做了哪些逻辑。

1．创建代理

```
public void afterPropertiesSet() {
   prepare();
}
public void prepare() {
    this.proxy = getProxyForService();
```

```
        }
        protected Object getProxyForService() {
            //验证service
            checkService();
            //验证serviceInterface
            checkServiceInterface();
            //使用JDK的方式创建代理
             ProxyFactory proxyFactory = new ProxyFactory();
            //添加代理接口
            proxyFactory.addInterface(getServiceInterface());
             if (this.registerTraceInterceptor != null ?
             this.registerTraceInterceptor.booleanValue() : this.interceptors ==
null) {
                //加入代理的横切面RemoteInvocationTraceInterceptor并记录
Exporter名称
                proxyFactory.addAdvice(new RemoteInvocationTraceInterceptor
(getExporterName()));
            }
             if (this.interceptors != null) {
                 AdvisorAdapterRegistry adapterRegistry =
GlobalAdvisorAdapterRegistry.
                getInstance();
                 for (int i = 0; i < this.interceptors.length; i++) {

proxyFactory.addAdvisor(adapterRegistry.wrap(this.interceptors[i]));
                }
            }
```

```
    //设置要代理的目标类
    proxyFactory.setTarget(getService());
    proxyFactory.setOpaque(true);
    //创建代理
     return proxyFactory.getProxy(getBeanClassLoader());
}
```

通过将上面3个方法串联，可以看到，初始化过程中实现的逻辑主要是创建了一个代理，代理中封装了对于特定请求的处理方法以及接口等信息，而这个代理的最关键目的是加入了RemoteInvocationTraceInterceptor增强器，当然创建代理还有些其他好处，比如代码优雅、方便扩展等。RemoteInvocationTraceInterceptor 中的增强主要是对增强的目标方法进行一些相关信息的日志打印，并没有在此基础上进行任何功能性的增强。那么这个代理究竟是在什么时候使用的呢？暂时留下悬念，我们接下来分析当有 Web 请求时 HttpRequestHandler 的 handleRequest方法的处理。

2．处理来自客户端的request

当有 Web 请求时，根据配置中的规则会把路径匹配的访问直接引入对应的 HttpRequest Handler 中。本例中的 Web 请求与普通的 Web 请求是有些区别的，因为此处的请求包含着HttpInvoker的处理过程。

```
public void handleRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    try {
      //从request中读取序列化对象
       RemoteInvocation invocation = readRemoteInvocation(request);
      //执行调用
       RemoteInvocationResult result =
```

invokeAndCreateResult(invocation, getProxy());
//将结果的序列化对象写入输出流
writeRemoteInvocationResult(request, response, result);
}
catch (ClassNotFoundException ex) {
throw new NestedServletException("Class not found during deserialization", ex);
}
}

在handlerRequest函数中，我们很清楚地看到了HttpInvoker处理的大致框架，HttpInvoker服务简单点说就是将请求的方法，也就是RemoteInvocation对象，从客户端序列化并通过Web请求出入服务端，服务端在对传过来的序列化对象进行反序列化还原RemoteInvocation实例，然后通过实例中的相关信息进行相关方法的调用，并将执行结果再次的返回给客户端。从handleRequest函数中我们也可以清晰地看到程序执行的框架结构。

（1）从request中读取序列化对象。

主要是从 HttpServletRequest 提取相关的信息，也就是提取HttpServletRequest 中的RemoteInvocation对象的序列化信息以及反序列化的过程。

```
protected RemoteInvocation readRemoteInvocation(HttpServletRequest request)
throws IOException, ClassNotFoundException {
    return readRemoteInvocation(request, request.getInputStream());
}
protected RemoteInvocation readRemoteInvocation(HttpServletRequest request, InputStream is)
```

```
        throws IOException, ClassNotFoundException {
            //创建对象输入流
            ObjectInputStream ois =
createObjectInputStream(decorateInputStream(request, is));
            try {
                //从输入流中读取序列化对象
                return doReadRemoteInvocation(ois);
            }
            finally {
                ois.close();
            }
        }
        protected RemoteInvocation
doReadRemoteInvocation(ObjectInputStream ois)
        throws IOException, ClassNotFoundException {
            Object obj = ois.readObject();
            if (!(obj instanceof RemoteInvocation)) {
                throw new RemoteException("Deserialized object needs to be
assignable to type [" +
                RemoteInvocation.class.getName() + "]: " + obj);
            }
            return (RemoteInvocation) obj;
        }
```

对于序列化提取与转换过程其实并没有太多需要解释的东西，这里完全是按照标准的方式进行操作，包括创建ObjectInputStream以及从ObjectInputStream中提取对象实例。

（2）执行调用。

根据反序列化方式得到的RemoteInvocation对象中的信息，进行方法调用。注意，此时调用的实体并不是服务接口或者服务类，而是之前在初始化时候构造的封装了服务接口以及服务类的代理。

完成了RemoteInvocation实例的提取，也就意味着可以通过RemoteInvocation实例中提供的信息进行方法调用了。

```
protected RemoteInvocationResult
invokeAndCreateResult(RemoteInvocation invocation,
    Object targetObject) {
        try {
          //激活代理类中对应invocation中的方法
           Object value = invoke(invocation, targetObject);
          //封装结果以便于序列化
           return new RemoteInvocationResult(value);
        }
        catch (Throwable ex) {
           return new RemoteInvocationResult(ex);
        }
    }
```

这段函数有两点需要说明的地方。

对应方法的激活也就是invoke方法的调用，虽然经过层层环绕，但是最终还是实现了一个我们熟知的调用invocation.invoke(targetObject)，也就是执行RemoteInvocation类中的 invoke 方法，大致的逻辑还是通过RemoteInvocation 中对应的方法信息在targetObject 上去执行，此方法在分析 RMI 功能的时候已经分析过，不再赘述。但是在对于当前方法的targetObject参数，此targetObject是代理类，调用代理类的时候需要考虑增强方法的调用，这是读者需要注意的地方。

对于返回结果需要使用 RemoteInvocationResult 进行封装，之所以

需要通过使用RemoteInvocationResult类进行封装，是因为无法保证对于所有操作的返回结果都继承Serializable接口，也就是说无法保证所有返回结果都可以直接进行序列化，那么，就必须使用RemoteInvocationResult类进行统一封装。

（3）将结果的序列化对象写入输出流。

同样这里也包括结果的序列化过程。

```
protected void writeRemoteInvocationResult(
HttpServletRequest request, HttpServletResponse response, RemoteInvocation
Result result) throws IOException {
   response.setContentType(getContentType());
   writeRemoteInvocationResult(request, response, result, response.getOutputStream());
}
protected void writeRemoteInvocationResult(
HttpServletRequest request, HttpServletResponse response, RemoteInvocation
Result result, OutputStream os)
throws IOException {
   //获取输入流
ObjectOutputStream oos = createObjectOutputStream(decorateOutputStream(request,
response, os));
    try {
      //将序列化对象写入输入流
      doWriteRemoteInvocationResult(result, oos);
   }
```

```
        finally {
            oos.close();
        }
    }
    protected void
doWriteRemoteInvocationResult(RemoteInvocationResult result,
ObjectOutput
    Stream oos)throws IOException {
        oos.writeObject(result);
    }
```

12.2.3 客户端实现

分析了服务端的解析以及处理过程后，我们接下来分析客户端的调用过程，在服务端调用的分析中我们反复提到需要从 HttpServletRequest 中提取从客户端传来的 RemoteInvocation 实例，然后进行相应解析。所以，在客户端，一个比较重要的任务就是构建RemoteInvocation实例，并传送到服务端。根据配置文件中的信息，我们还是首先锁定HttpInvokerProxyFactoryBean类，并查看其层次结构，如图12-4所示。

从层次结构中我们看到，HttpInvokerProxyFactoryBean类同样实现了InitializingBean接口。同时，又实现了 FactoryBean 以及 MethodInterceptor。这已经是老生常谈的问题了，实现这几个接口以及这几个接口在 Spring 中会有什么作用就不再赘述了，我们还是根据实现的InitializingBean接口分析初始化过程中的逻辑。

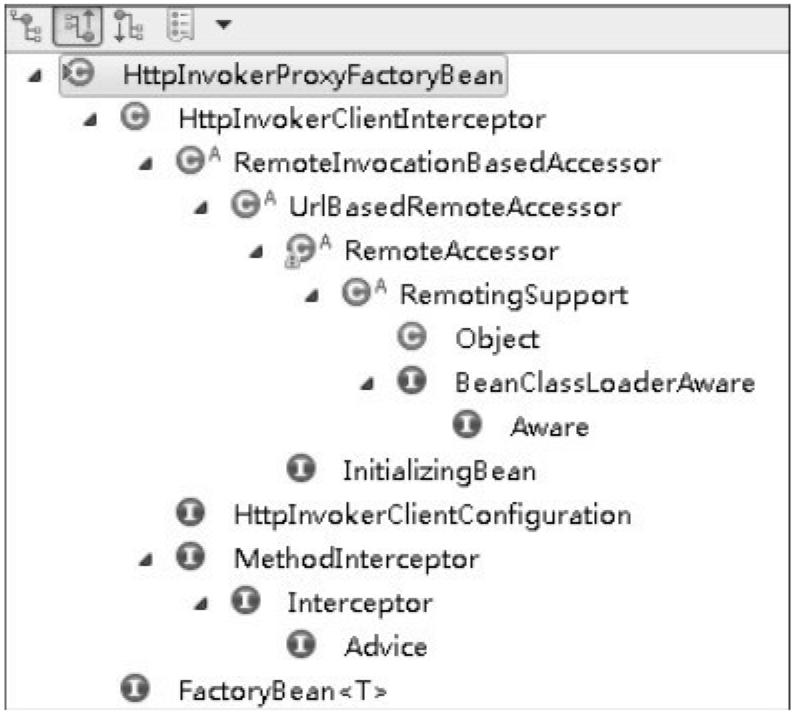


图12-4 HttpInvokerProxyFactoryBean类的层次结构图

```
public void afterPropertiesSet() {
    super.afterPropertiesSet();
     if (getServiceInterface() == null) {
        throw new IllegalArgumentException("Property 'serviceInterface'
is
        required");
    }
    //创建代理并使用当前方法为拦截器增强
     this.serviceProxy = new ProxyFactory(getServiceInterface(),
this).getProxy
    (getBeanClassLoader());
}
```

在afterPropertiesSet中主要创建了一个代理，该代理封装了配置的服

务接口，并使用当前类也就是 HttpInvokerProxyFactoryBean 作为增强。因为 HttpInvokerProxyFactoryBean 实现了MethodInterceptor方法，所以可以作为增强拦截器。

同样，又由于HttpInvokerProxyFactoryBean实现了FactoryBean接口，所以通过Spring中普通方式调用该bean时调用的并不是该bean本身，而是此类中getObject方法返回的实例，也就是实例化过程中所创建的代理。

```
public Object getObject() {
    return this.serviceProxy;
}
```

那么，综合之前的使用示例，我们再次回顾一下，HttpInvokerProxyFactoryBean类型bean在初始化过程中创建了封装服务接口的代理，并使用自身作为增强拦截器，然后又因为实现了FactoryBean接口，所以获取Bean的时候返回的其实是创建的代理。那么，汇总上面的逻辑，当调用如下代码时，其实是调用代理类中的服务方法，而在调用代理类中的服务方法时又会使用代理类中加入的增强器进行增强。

```
ApplicationContext context = new ClassPathXmlApplicationContext("classpath:client.xml");
HttpInvokerTestI httpInvokerTestI = (HttpInvokerTestI) context.getBean("remoteService");
System.out.println(httpInvokerTestI.getTestPo("dddd"));
```

这时，所有的逻辑分析其实已经被转向了对于增强器也就是HttpInvokerProxyFactoryBean类本身的invoke方法的分析。

在分析invoke方法之前，其实我们已经猜出了该方法所提供的主要功能就是将调用信息封装在RemoteInvocation中，发送给服务端并等待返回结果。

```
    public Object invoke(MethodInvocation methodInvocation) throws
Throwable {
        if (AopUtils.isToStringMethod(methodInvocation.getMethod())) {
            return "HTTP invoker proxy for service URL [" + getServiceUrl()
+ "]";
        }
        //将要调用的方法封装为RemoteInvocation
        RemoteInvocation invocation =
createRemoteInvocation(methodInvocation);
        RemoteInvocationResult result = null;
        try {
          //远程执行方法
            result = executeRequest(invocation, methodInvocation);
        }
        catch (Throwable ex) {
            throw convertHttpInvokerAccessException(ex);
        }
        try {
          //提取结果
            return recreateRemoteInvocationResult(result);
        }
        catch (Throwable ex) {
            if (result.hasInvocationTargetException()) {
                throw ex;
            }
            else {
                throw new RemoteInvocationFailureException("Invocation of
```

```
method [" +
            methodInvocation.getMethod() +
            "] failed in HTTP invoker remote service at [" + getServiceUrl()
            + "]", ex);
        }
      }
    }
```

函数主要有3个步骤。

（1）构建RemoteInvocation实例。

因为是代理中增强方法的调用，调用的方法及参数信息会在代理中封装至 MethodInvocation实例中，并在增强方器中进行传递，也就意味着当程序进入invoke方法时其实是已经包含了调用的接口的相关信息的，那么，首先要做的就是将 MethodInvocation 中的信息提取并构建 RemoteInvocation实例。

（2）远程执行方法。

（3）提取结果。

考虑到序列化的问题，在Spring中约定使用HttpInvoker方式进行远程方法调用时，结果使用 RemoteInvocationResult 进行封装，那么在提取结果后还需要从封装的结果中提取对应的结果。

而在这三个步骤中最为关键的就是远程方法的执行。执行远程调用的首要步骤就是将调用方法的实例写入输出流中。

```
    protected RemoteInvocationResult executeRequest(
    RemoteInvocation invocation, MethodInvocation originalInvocation)
throws Exception {
      return executeRequest(invocation);
    }
    protected RemoteInvocationResult executeRequest(RemoteInvocation
```

```
invocation) throws Exception {
        return getHttpInvokerRequestExecutor().executeRequest(this,
invocation);
    }
    public final RemoteInvocationResult executeRequest(
    HttpInvokerClientConfiguration config, RemoteInvocation invocation)
throws
    Exception {
    //获取输出流
        ByteArrayOutputStream baos =
getByteArrayOutputStream(invocation);
        if (logger.isDebugEnabled()) {
            logger.debug("Sending HTTP invoker request for service at [" +
            config.getServiceUrl() +

"], with size " + baos.size());

        }
        return doExecuteRequest(config, baos);
    }
```

在doExecuteRequest方法中真正实现了对远程方法的构造与通信，与远程方法的连接功能实现中，Spring引入了第三方JAR：HttpClient。HttpClient是Apache Jakarta Common下的子项目，可以用来提供高效的、最新的、功能丰富的支持HTTP协议的客户端编程工具包，并且它支持HTTP协议最新的版本和建议。对HttpClient的具体使用方法有兴趣的读者可以参考更多的资料和文档。

```
    protected RemoteInvocationResult doExecuteRequest(
```

```
HttpInvokerClientConfiguration config, ByteArrayOutputStream baos)
throws IOException, ClassNotFoundException {
    //创建HttpPost
     HttpPost postMethod = createHttpPost(config);
    //设置含有方法的输出流到post中
     setRequestBody(config, postMethod, baos);
     try {
        //执行方法并等待结果响应
         HttpResponse response = executeHttpPost(config, getHttpClient(),
postMethod);
        //验证
         validateResponse(config, response);
        //提取返回的输入流
         InputStream responseBody = getResponseBody(config, response);
        //从输入流中提取结果
         return readRemoteInvocationResult(responseBody,
config.getCodebaseUrl());
    }
     finally {
         if (releaseConnectionMethod != null){
             ReflectionUtils.invokeMethod(releaseConnectionMethod,
postMethod);
        }
    }
}
```

接下来我们逐步分析客户端实现的逻辑。

1．创建**HttpPost**

由于对于服务端方法的调用是通过Post方式进行的，那么首先要做的就是构建HttpPost，构建HttpPost过程中可以设置一些必要的参数。

```
protected PostMethod
createPostMethod(HttpInvokerClientConfiguration config) throws
IOException {
   //设置需要访问的url
    PostMethod postMethod = new PostMethod(config.getServiceUrl());
    LocaleContext locale = LocaleContextHolder.getLocaleContext();
    if (locale != null) {
      //加入Accept-Language属性

postMethod.addRequestHeader(HTTP_HEADER_ACCEPT_LANGUAGE,
StringUtils.
      toLanguageTag(locale.getLocale()));
   }
    if (isAcceptGzipEncoding()) {
      //加入Accept-Encoding属性

postMethod.addRequestHeader(HTTP_HEADER_ACCEPT_ENCODING,
ENCODING_GZIP);
   }
    return postMethod;
}
```

2．设置**RequestBody**

构建好PostMethod实例后便可以将存储RemoteInvocation实例的序列化对象的输出流设置进去，当然这里需要注意的是传入的ContentType类型，一定要传入application/x-java-serialized-object以保证

服务端解析时会按照序列化对象的解析方式进行解析。

protected void setRequestBody(

HttpInvokerClientConfiguration config, PostMethod postMethod, ByteArrayOutputStream

baos)

throws IOException {

//将序列化流加入到postMethod中并声明ContentType类型为 application/x-java-serialized-object

postMethod.setRequestEntity(new ByteArrayRequestEntity(baos.toByteArray(),

getContentType()));

}

3．执行远程方法

通过HttpClient所提供的方法来直接执行远程方法。

protected void executePostMethod(

HttpInvokerClientConfiguration config, HttpClient httpClient, PostMethod

postMethod) throws IOException {

httpClient.executeMethod(postMethod);

}

4．远程相应验证

对于HTTP调用的响应码处理，大于300则是非正常调用的响应码。

protected void validateResponse(HttpInvokerClientConfiguration config, PostMethod

postMethod)

throws IOException {

if (postMethod.getStatusCode() >= 300) {

```
throw new HttpException(
"Did not receive successful HTTP response: status code = " +
postMethod.getStatusCode() +
", status message = [" + postMethod.getStatusText() + "]");
}
}
```

5．提取响应信息

从服务器返回的输入流可能是经过压缩的，不同的方式采用不同的办法进行提前。

```
protected InputStream
getResponseBody(HttpInvokerClientConfiguration config, PostMethod
postMethod)
throws IOException {
    if (isGzipResponse(postMethod)) {
        return new
GZIPInputStream(postMethod.getResponseBodyAsStream());
    }
    else {
        return postMethod.getResponseBodyAsStream();
    }
}
```

6．提取返回结果

提取结果的流程主要是从输入流中提取响应的序列化信息。

```
protected RemoteInvocationResult
readRemoteInvocationResult(InputStream is, String
```

```
codebaseUrl)
throws IOException, ClassNotFoundException {
    ObjectInputStream ois =
createObjectInputStream(decorateInputStream(is),
   codebaseUrl);
    try {
        return doReadRemoteInvocationResult(ois);
    }
    finally {
       ois.close();
    }
}
```

第13章　Spring消息

Java消息服务（Java Message Service，JMS）应用程序接口是一个Java平台中关于面向消息中间件（MOM）的API，用于在两个应用程序之间或分布式系统中发送消息，进行异步通信。Java消息服务是一个与具体平台无关的API，绝大多数MOM提供商都对JMS提供支持。

Java消息服务的规范包括两种消息模式，点对点和发布者/订阅者。许多提供商支持这一通用框架。因此，程序员可以在他们的分布式软件中实现面向消息的操作，这些操作将具有不同面向消息中间件产品的可移植性。

Java消息服务支持同步和异步的消息处理，在某些场景下，异步消息是必要的，而且比同步消息操作更加便利。

Java消息服务支持面向事件的方法接收消息，事件驱动的程序设计现在被广泛认为是一种富有成效的程序设计范例，程序员们都相当熟

悉。

在应用系统开发时，Java消息服务可以推迟选择面对消息中间件产品，也可以在不同的面对消息中间件切换。

本章以Java消息服务的开源实现产品ActiveMQ为例来进行Spring整合消息服务功能的实现分析。

13.1 JMS的独立使用

尽管大多数的Java消息服务的使用都会跟Spring相结合，但是，我们还是非常有必要了解消息的独立使用方法，这对于我们了解消息的实现原理以及后续的与 Spring 整合实现分析都非常重要。当然在消息服务的使用前，需要我们先开启消息服务器，如果是Windows系统下可以直接双击ActiveMQ安装目录下的bin目录下的activemq.bat文件来启动消息服务器。

消息服务的使用除了要开启消息服务器外，还需要构建消息的发送端与接收端，发送端主要用来将包含业务逻辑的消息发送至消息服务器，而消息接收端则用于将服务器中的消息提取并进行相应的处理。

（1）发送端实现

发送端主要用于发送消息到消息服务器，以下为发送消息测试，尝试发送三条消息到消息服务器，消息的内容为“大家好这是个测试”。

```
public class Sender {
    public static void main(String[] args) throws Exception {
        ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory();
        Connection connection = connectionFactory.createConnection();
        //connection.start();
        Session session = connection.createSession(Boolean.TRUE,
```

```
Session.AUTO_ACKNOWLEDGE);
 Destination destination = session.createQueue("my-queue");
 MessageProducer producer = session.createProducer(destination);
 for (int i = 0; i < 3; i++) {
     TextMessage message = session.createTextMessage("大家好这是个测试");
    Thread.sleep(1000);
    // 通过消息生产者发出消息
    producer.send(message);
}
session.commit();
session.close();
connection.close();
}
}
```

上面的函数实现很容易让我们联想到数据库的实现，在函数开始时需要一系列冗余的但又必不可少的用于连接的代码，而其中真正用于发送消息的代码其实很简单。

（2）接收端实现。

接收端主要用于连接消息服务器并接收服务器上的消息。

```
public class Receiver {
    public static void main(String[] args) throws Exception {
         ConnectionFactory connectionFactory = new ActiveMQConnectionFactory();
         Connection connection = connectionFactory.createConnection();
        connection.start();
         final Session session = connection.createSession(Boolean.TRUE,
```

```
Session.AUTO_
                ACKNOWLEDGE);
                 Destination destination = session.createQueue("my-queue");
                 MessageConsumer consumer =
session.createConsumer(destination);
                 int i=0;
                 while(i<3) {
                    i++;
                     TextMessage message = (TextMessage) consumer.receive();
                    session.commit();
                     //TODO something....
                     System.out.println("收到消息：" + message.getText());
                }
                session.close();
                connection.close();
            }
        }
```

程序测试的顺序是首先开启发送端，然后向服务器发送消息，接着再开启接收端，不出意外，就会接收到发送端发出的消息。

13.2 Spring整合ActiveMQ

整个消息的发送与接收过程非常简单，但是其中却参杂着大量的冗余代码，比如Connection的创建与关闭，Session的创建与关闭等，为了消除这一冗余工作量，Spring进行了进一步的封装。Spring下的ActiveMQ使用方式如下。

（1）Spring配置文件。

配置文件是Spring 的核心，Spring 整合消息服务的使用也从配置文件配置开始。类似于数据库操作，Spring也将ActiveMQ中的操作统一封装至JmsTemplate中，以方便我们统一使用。所以，在 Spring 的核心配置文件中首先要注册 JmsTemplate 类型的 bean。当然，ActiveMQConnection Factory 用于连接消息服务器，是消息服务的基础，也要注册。ActiveMQQueue 则用于指定消息的目的地。

```
<beans>
    <bean id="connectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL">
            <value>tcp://localhost:61616</value>
        </property>
    </bean>
    <bean id="jmsTemplate"
class="org.Springframework.jms.core.JmsTemplate">
        <property name="connectionFactory">
            <ref bean="connectionFactory" />
        </property>
    </bean>
    <bean id="destination"
class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg index="0">
            <value>HelloWorldQueue</value>
        </constructor-arg>
    </bean>
</beans>
```

（2）发送端。

有了以上的配置，Spring就可以根据配置信息简化我们的工作量。Spring中使用发送消息到消息服务器，省去了冗余的Connection以及Session等的创建与销毁过程，简化了工作量。

```
public class HelloWorldSender {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new
ClassPathXmlApplicationContext(
        new String[] { "test/activeMQ/Spring/applicationContext.xml" });
        JmsTemplate jmsTemplate = (JmsTemplate)
context.getBean("jmsTemplate");
        Destination destination = (Destination)
context.getBean("destination");
        jmsTemplate.send(destination, new MessageCreator() {
            public Message createMessage(Session session) throws
JMSException {
                return session.createTextMessage("大家好这个是测试！");
            }
        });
    }
}
```

（3）接收端。

同样，在Spring中接收消息也非常方便，Spring中连接服务器接收消息的示例如下：

```
public class HelloWorldReciver {
    public static void main(String args[]) throws Exception {
        ApplicationContext context = new
ClassPathXmlApplicationContext(
```

```
            new String[] { "test/activeMQ/Spring/applicationContext.xml" });
            JmsTemplate jmsTemplate = (JmsTemplate)
context.getBean("jmsTemplate");
            Destination destination = (Destination)
context.getBean("destination");
            TextMessage msg = (TextMessage)
jmsTemplate.receive(destination);
            System.out.println("reviced msg is:" + msg.getText());
        }
    }
```

到这里我们已经完成了Spring消息的发送与接收操作。但是，如HelloWorldReciver中所示的代码，使用 jmsTemplate.receive(destination)方法只能接收一次消息，如果未接收到消息，则会一直等待，当然用户可以通过设置 timeout 属性来控制等待时间，但是一旦接收到消息本次接收任务就会结束，虽然用户可以通过 while(true)的方式来实现循环监听消息服务器上的消息，还有一种更好的解决办法：创建消息监听器。消息监听器的使用方式如下。

（1）创建消息监听器。

用于监听消息，一旦有新消息Spring会将消息引导至消息监听器以方便用户进行相应的逻辑处理。

```
public class MyMessageListener implements MessageListener{
    @Override
    public void onMessage(Message arg0) {
        TextMessage msg = (TextMessage) arg0;
        try {
            System.out.println(msg.getText());
        } catch (JMSException e) {
```

```
                e.printStackTrace();
            }
        }
    }
```

（2）修改配置文件。

为了使用消息监听器，需要在配置文件中注册消息容器，并将消息监听器注入到容器中。

```
<beans>
    <bean id="connectionFactory"
class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL">
            <value>tcp://localhost:61616</value>
        </property>
    </bean>
    <bean id="jmsTemplate"
class="org.Springframework.jms.core.JmsTemplate">
        <property name="connectionFactory">
            <ref bean="connectionFactory" />
        </property>
    </bean>
    <bean id="destination"
class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg index="0">
            <value>HelloWorldQueue</value>
        </constructor-arg>
    </bean>
    <bean id="myTextListener"
```

```
class="test.activeMQ.Spring.MyMessageListener" />
    <bean id="javaConsumer"

class="org.Springframework.jms.listener.DefaultMessageListenerContainer">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="destination" ref="destination" />
        <property name="messageListener" ref="myTextListener" />
    </bean>
</beans>
```

通过以上的修改便可以进行消息监听的功能了，一旦有消息传入至消息服务器，则会被消息监听器监听到，并由Spring将消息内容引导至消息监听器的处理函数中等待用户的进一步逻辑处理。

13.3 源码分析

尽管消息接收可以使用消息监听器的方式替代模版方法，但是在发送的时候是无法替代的，在Spring中必须要使用JmsTemplate提供的方法来进行发送操作，可见JmsTemplate类的重要性，那么我们对于Spring整合消息服务的分析就从JmsTemplate开始。

13.3.1 JmsTemplate

在代码与Spring整合的实例中，我们看到Spring采用了与JDBC等一贯的套路，为我们提供了JmsTemplate来封装常用操作。查看JmsTemplate的类型层级结构图，如图13-1所示。



图13-1 JmsTemplate的类型层级结构图

首先还是按照一贯的分析套路，提取我们感兴趣的接口InitializingBean，接口方法实现是在JmsAccessor类中，如下：

```
public void afterPropertiesSet() {
    if (getConnectionFactory() == null) {
        throw new IllegalArgumentException("Property 'connectionFactory' is required");
    }
}
```

发现函数中只是一个验证的功能，并没有逻辑实现。丢掉这个线索，我们转向实例代码的分析。首先以发送为例，在Spring中发送消息可以通过JmsTemplate中提供的方法来实现。

```
public void send(final Destination destination, final MessageCreator messageCreator)
throws JmsException
```

使用方式如下：

```
jmsTemplate.send(destination, new MessageCreator() {
    public Message createMessage(Session session) throws JMSException {
```

```
            return session.createTextMessage("大家好这个是测试！");
        }
    });
```

我们就跟着程序流，进入函数send查看其源代码：

```
public void send(final Destination destination, final MessageCreator messageCreator)
throws JmsException {
    execute(new SessionCallback<Object>() {
        public Object doInJms(Session session) throws JMSException {
            doSend(session, destination, messageCreator);
            return null;
        }
    }, false);
}
```

现在的风格不得不让我们回想起JdbcTemplate的类实现风格，极为相似，都是提取一个公共的方法作为最底层、最通用的功能实现，然后又通过回调函数的不同来区分个性化的功能。我们首先查看通用代码的抽取实现。

1．通用代码抽取

根据之前分析JdbcTemplate的经验，我们推断，在execute中一定是封装了Connection以及Session的创建操作。

```
public <T> T execute(SessionCallback<T> action, boolean startConnection) throws JmsException {
      Assert.notNull(action, "Callback object must not be null");
      Connection conToClose = null;
      Session sessionToClose = null;
      try {
```

```
            Session sessionToUse =
ConnectionFactoryUtils.doGetTransactionalSession(
            getConnectionFactory(), this.transactionalResourceFactory,
           startConnection);
            if (sessionToUse == null) {
               //创建connection
               conToClose = createConnection();
               //根据connection创建session
               sessionToClose = createSession(conToClose);
               //是否开启向服务器推送连接信息，只有接收信息时需要，
发送时不需要
               if (startConnection) {
                  conToClose.start();
               }
               sessionToUse = sessionToClose;
            }
            if (logger.isDebugEnabled()) {
               logger.debug("Executing callback on JMS Session: " +
sessionToUse);
            }
            //调用回调函数
            return action.doInJms(sessionToUse);
         }
         catch (JMSException ex) {
            throw convertJmsAccessException(ex);
         }
         finally {
```

```
            //关闭session
            JmsUtils.closeSession(sessionToClose);
            //释放连接
             ConnectionFactoryUtils.releaseConnection(conToClose,
getConnectionFactory(),
            startConnection);
        }
    }
```

在展示单独使用 activeMQ 时，我们知道为了发送一条消息需要做很多工作，需要很多的辅助代码，而这些代码又都是千篇一律的，没有任何的差异，所以 execute 方法的目的就是帮助我们抽离这些冗余代码使我们更加专注于业务逻辑的实现。从函数中看，这些冗余代码包括创建Connection、创建Session、当然也包括关闭Session和关闭Connection。而在准备工作结束后，调用回调函数将程序引入用户自定义实现的个性化处理。至于如何创建 Session 与Connection，有兴趣的读者可以进一步研究Mybatis的源码。

2．发送消息的实现

有了基类辅助实现，使Spring更加专注于个性的处理，也就是说Spring使用execute方法中封装了冗余代码，而将个性化的代码实现放在了回调函数doInJms函数中。在发送消息的功能中回调函数通过局部类实现。

```
new SessionCallback<Object>() {
    public Object doInJms(Session session) throws JMSException {
        doSend(session, destination, messageCreator);
        return null;
    }
}
```

此时的发送逻辑已经完全被转向了doSend方法，这样使整个功能实现变得更加清晰。

```
protected void doSend(Session session, Destination destination,
MessageCreator
    messageCreator)
    throws JMSException {
        Assert.notNull(messageCreator, "MessageCreator must not be null");
        MessageProducer producer = createProducer(session, destination);
        try {
            Message message = messageCreator.createMessage(session);
            if (logger.isDebugEnabled()) {
                logger.debug("Sending created message: " + message);
            }
            doSend(producer, message);
            // Check commit - avoid commit call within a JTA transaction.
            if (session.getTransacted() &&
isSessionLocallyTransacted(session)) {
                // Transacted session created by this template -> commit.
                JmsUtils.commitIfNecessary(session);
            }
        }
        finally {
            JmsUtils.closeMessageProducer(producer);
        }
    }
    protected void doSend(MessageProducer producer, Message message)
throws JMSException {
```

```
            if (isExplicitQosEnabled()) {
                producer.send(message, getDeliveryMode(), getPriority(),
getTimeToLive());
            }
            else {
               producer.send(message);
            }
        }
```

在演示独立使用消息功能的时候，我们大体了解了消息发送的基本套路，虽然这些步骤已经被Spring拆得支离破碎，但是我们还是能捕捉到一些影子。在发送消息还是遵循着消息发送的规则，比如根据Destination创建MessageProducer、创建Message，并使用MessageProducer实例来发送消息。

3．接收消息

我们通常使用jmsTemplate.receive(destination)来接收简单的消息，那么这个功能Spring是如何封装的呢？

```
        public Message receive(Destination destination) throws JmsException {
             return receiveSelected(destination, null);
        }
        public Message receiveSelected(final Destination destination, final
String messageSelector)
        throws JmsException {
             return execute(new SessionCallback<Message>() {
                 public Message doInJms(Session session) throws JMSException {
                     return doReceive(session, destination, messageSelector);
                 }
             }, true);
```

```
}
protected Message doReceive(Session session, Destination destination,
String messageSelector)
throws JMSException {
    return doReceive(session, createConsumer(session, destination,
messageSelector));
}
protected Message doReceive(Session session, MessageConsumer
consumer) throws JMSException {
    try {
        // Use transaction timeout (if available).
        long timeout = getReceiveTimeout();
        JmsResourceHolder resourceHolder =
        (JmsResourceHolder)
TransactionSynchronizationManager.getResource
        (getConnectionFactory());
        if (resourceHolder != null && resourceHolder.hasTimeout()) {
            timeout = Math.min(timeout,
resourceHolder.getTimeToLiveInMillis());
        }
        Message message = doReceive(consumer, timeout);
        if (session.getTransacted()) {
            // Commit necessary - but avoid commit call within a JTA
transaction.
            if (isSessionLocallyTransacted(session)) {
                // Transacted session created by this template -> commit.
                JmsUtils.commitIfNecessary(session);
```

```
                }
            }
            else if (isClientAcknowledge(session)) {
                // Manually acknowledge message, if any.
                if (message != null) {
                    message.acknowledge();
                }
            }
            return message;
        }
        finally {
            JmsUtils.closeMessageConsumer(consumer);
        }
    }
    private Message doReceive(MessageConsumer consumer, long timeout)
throws JMSException {
        if (timeout == RECEIVE_TIMEOUT_NO_WAIT) {
            return consumer.receiveNoWait();
        }
        else if (timeout > 0) {
            return consumer.receive(timeout);
        }
        else {
            return consumer.receive();
        }
    }
```

实现的套路与发送差不多，同样还是使用 execute 函数来封装冗余

的公共操作，而最终的目标还是通过consumer.receive()来接收消息，其中的过程就是对于MessageConsumer的创建以及一些辅助操作。

13.3.2 监听器容器

消息监听器容器是一个用于查看 JMS 目标等待消息到达的特殊bean，一旦消息到达它就可以获取到消息，并通过调用onMessage()方法将消息传递给一个MessageListener实现。Spring中消息监听器容器的类型如下。

SimpleMessageListenerContainer：最简单的消息监听器容器，只能处理固定数量的JMS会话，且不支持事务。

DefaultMessageListenerContainer：这个消息监听器容器建立在SimpleMessageListener Container容器之上，添加了对事务的支持。

serversession.ServerSessionMessage.ListenerContainer：这是功能最强大的消息监听器，与DefaultMessageListenerContainer相同，它支持事务，但是它还允许动态地管理JMS会话。

下面以 DefaultMessageListenerContainer 为例进行分析，看看消息监听器容器的实现。在之前消息监听器的使用示例中，我们了解到在使用消息监听器容器时一定要将自定义的消息监听器置入到容器中，这样才可以在收到信息时，容器把消息转向监听器处理。查看DefaultMessageListenerContainer层次结构图，如图13-2所示。

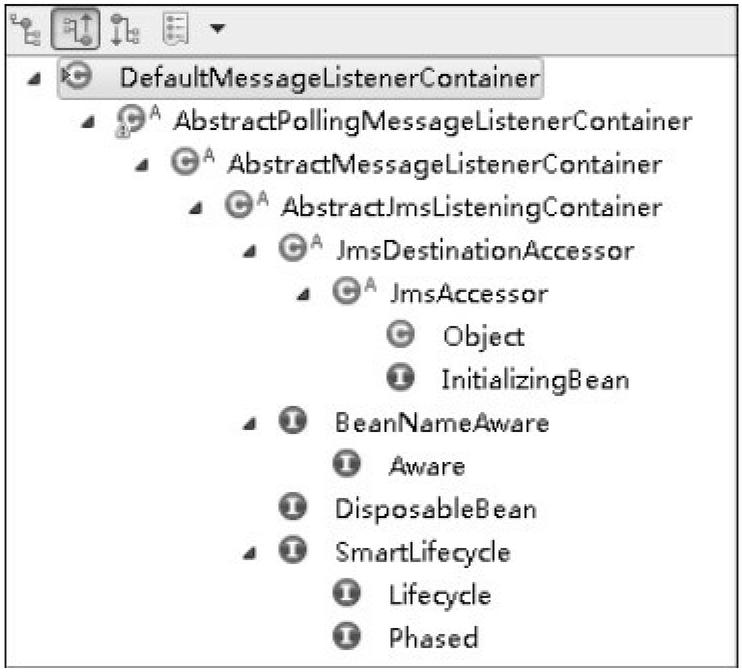


图13-2 DefaultMessageListenerContainer层次结构图

同样，我们看到此类实现了InitializingBean接口，按照以往的风格我们还是首先查看接口方法afterPropertiesSet()中的逻辑，其方法实现在其父类AbstractJmsListeningContainer中。

```
public void afterPropertiesSet() {
    //验证connectionFactory
    super.afterPropertiesSet();
    //验证配置文件
    validateConfiguration();
    //初始化
    initialize();
}
```

监听器容器的初始化只包含了三句代码，其中前两句只用于属性的验证，比如connectionFacory或者destination等属性是否为空等，而真正用于初始化的操作委托在initialize()中执行。

```
public void initialize() throws JmsException {
    try {
        //lifecycleMonitor用于控制生命周期的同步处理
        synchronized (this.lifecycleMonitor) {
            this.active = true;
            this.lifecycleMonitor.notifyAll();
        }
        doInitialize();
    }
    catch (JMSException ex) {
        synchronized (this.sharedConnectionMonitor) {

ConnectionFactoryUtils.releaseConnection(this.sharedConnection,
                getConnectionFactory(), this.autoStartup);
                this.sharedConnection = null;
        }
        throw convertJmsAccessException(ex);
    }
}
protected void doInitialize() throws JMSException {
    synchronized (this.lifecycleMonitor) {
        for (int i = 0; i < this.concurrentConsumers; i++) {
            scheduleNewInvoker();
        }
    }
}
```

这里用到了concurrentConsumers属性，网络中对此属性用法的说明

如下。

消息监听器允许创建多个 Session 和 MessageConsumer 来接收消息。具体的个数由concurrentConsumers属性指定。需要注意的是，应该只是在Destination为Queue的时候才使用多个 MessageConsumer（Queue 中的一个消息只能被一个 Consumer 接收），虽然使用多个MessageConsumer会提高消息处理的性能，但是消息处理的顺序却得不到保证。消息被接收的顺序仍然是消息发送时的顺序，但是由于消息可能会被并发处理，因此消息处理的顺序可能和消息发送的顺序不同。此外，不应该在Destination为Topic的时候使用多个MessageConsumer，因为多个MessageConsumer会接收到同样的消息。

对于具体的实现逻辑我们只能继续查看源码：

```
private void scheduleNewInvoker() {
    AsyncMessageListenerInvoker invoker = new
AsyncMessageListenerInvoker();
    if (rescheduleTaskIfNecessary(invoker)) {
// This should always be true, since we're only calling this when active.
this.scheduledInvokers.add(invoker);
   }
}
protected final boolean rescheduleTaskIfNecessary(Object task) {
    if (this.running) {
       try {
          doRescheduleTask(task);
       }
       catch (RuntimeException ex) {
          logRejectedTask(task, ex);
          this.pausedTasks.add(task);
```

```
            }
            return true;
        }
        else if (this.active) {
            this.pausedTasks.add(task);
            return true;
        }
        else {
            return false;
        }
    }
    protected void doRescheduleTask(Object task) {
        this.taskExecutor.execute((Runnable) task);
    }
```

分析源码得知，根据concurrentConsumers数量建立了对应数量的线程，即使读者不了解线程池的使用，至少根据以上代码可以推断出doRescheduleTask函数其实是在开启一个线程执行Runnable。我们反追踪这个传入的参数，可以看到这个参数其实是AsyncMessageListenerInvoker类型实例。因此我们可以推断，Spring根据concurrentConsumers数量建立了对应数量的线程，而每个线程都作为一个独立的接收者在循环接收消息。

于是我们把所有的焦点转向AsyncMessageListenerInvoker这个类的实现，由于它是作为一个Runnable角色去执行，所以对以这个类的分析从run方法开始。

```
    public void run() {
        //并发控制
        synchronized (lifecycleMonitor) {
```

```
            activeInvokerCount++;
            lifecycleMonitor.notifyAll();
        }
         boolean messageReceived = false;
         try {
            //根据每个任务设置的最大处理消息数量而作不同处理
            //小于0默认为无限制，一直接收消息
             if (maxMessagesPerTask < 0) {
                 messageReceived = executeOngoingLoop();
            }
             else {
                 int messageCount = 0;
                //消息数量控制，一旦超出数量则停止循环
                 while (isRunning() && messageCount < maxMessagesPerTask)
{
                     messageReceived = (invokeListener() || messageReceived);
                    messageCount++;
                }
            }
        }
         catch (Throwable ex) {
            //清理操作，包括关闭session等
            clearResources();
             if (!this.lastMessageSucceeded) {
                 // We failed more than once in a row - sleep for recovery
interval
                 // even before first recovery attempt.
```

```
        sleepInbetweenRecoveryAttempts();
    }
    this.lastMessageSucceeded = false;
    boolean alreadyRecovered = false;
    synchronized (recoveryMonitor) {
        if (this.lastRecoveryMarker == currentRecoveryMarker) {
            handleListenerSetupFailure(ex, false);
            recoverAfterListenerSetupFailure();
            currentRecoveryMarker = new Object();
        }
        else {
            alreadyRecovered = true;
        }
    }
    if (alreadyRecovered) {
        handleListenerSetupFailure(ex, true);
    }
}
finally {
    synchronized (lifecycleMonitor) {
        decreaseActiveInvokerCount();
        lifecycleMonitor.notifyAll();
    }
    if (!messageReceived) {
        this.idleTaskExecutionCount++;
    }
    else {
```

```
            this.idleTaskExecutionCount = 0;
        }
        synchronized (lifecycleMonitor) {
            if (!shouldRescheduleInvoker(this.idleTaskExecutionCount) ||
!reschedule
            TaskIfNecessary(this)) {
                // We're shutting down completely.
                scheduledInvokers.remove(this);
                if (logger.isDebugEnabled()) {
                    logger.debug("Lowered scheduled invoker count: " +
                    scheduledInvokers.size());
                }
                lifecycleMonitor.notifyAll();
                clearResources();
            }
            else if (isRunning()) {
                int nonPausedConsumers = getScheduledConsumerCount() -
                getPausedTaskCount();
                if (nonPausedConsumers < 1) {
                    logger.error("All scheduled consumers have been paused,
                    probably due to tasks having been rejected. " +
                    "Check your thread pool configuration! Manual
                    recovery necessary through a start() call.");
                }
                else if (nonPausedConsumers < getConcurrentConsumers()) {
                    logger.warn("Number of scheduled consumers has dropped
                    below concurrentConsumers limit, probably " +
```

```
                "due to tasks having been rejected. Check your
                thread pool configuration! Automatic recovery " +
                "to be triggered by remaining consumers.");
            }
        }
    }
  }
}
```

以上函数中主要根据变量maxMessagesPerTask的值来分为不同的情况处理，当然，函数中还使用了大量的代码处理异常机制的数据维护，但是我相信大家跟我一样更加关注程序的正常流程是如何处理的。

其实核心的处理就是调用 invokeListener 来接收消息并激活消息监听器，但是之所以两种情况分开处理，正是考虑到在无限制循环接收消息的情况下，用户可以通过设置标志位running来控制消息接收的暂停与恢复，并维护当前消息监听器的数量。

```
private boolean executeOngoingLoop() throws JMSException {
    boolean messageReceived = false;
    boolean active = true;
    while (active) {
        synchronized (lifecycleMonitor) {
            boolean interrupted = false;
            boolean wasWaiting = false;
            //如果当前任务已经处于激活状态但是却给了暂时终止的命令
            while ((active = isActive()) && !isRunning()) {
                if (interrupted) {
                    throw new IllegalStateException("Thread was interrupted
```

```
        while waiting for " +
        "a restart of the listener container, but
        container is still stopped");
    }
    if (!wasWaiting) {
       //如果并非处于等待状态则说明是第一次执行，需要将
激活任务数量减少
       decreaseActiveInvokerCount();
    }
    //开始进入等待状态，等待任务的恢复命令
    wasWaiting = true;
    try {
       //通过wait等待，也就是等待notify或者notifyAll
       lifecycleMonitor.wait();
    }
    catch (InterruptedException ex) {
        // Re-interrupt current thread, to allow other threads
        to react.
       Thread.currentThread().interrupt();
        interrupted = true;
    }
}
if (wasWaiting) {
    activeInvokerCount++;
}
if (scheduledInvokers.size() > maxConcurrentConsumers) {
    active = false;
```

```
            }
        }
        //正常处理流程
         if (active) {
            messageReceived = (invokeListener() || messageReceived);
        }
    }
     return messageReceived;
}
```

如果按照正常的流程其实是不会进入while循环中的，而是直接进入函数invokeListener()来接收消息并激活监听器，但是，我们不可能让循环一直持续下去，我们要考虑到暂停线程或者恢复线程的情况，这时，isRunning()函数就派上用场了。

isRunning()用来检测标志位this.running状态进而判断是否需要进入while循环。由于要维护当前线程激活数量，所以引入了wasWaiting变量，用来判断线程是否处于等待状态。如果线程首次进入等待状态，则需要减少线程激活数量计数器。

当然，还有一个地方需要提一下，就是线程等待不是一味地采用while循环来控制，因为如果单纯地采用while循环会浪费CPU的始终周期，给资源造成巨大的浪费。这里，Spring采用的是使用全局控制变量lifecycleMonitor的wait()方法来暂停线程，所以，如果终止线程需要再次恢复的话，除了更改this.running标志位外，还需要调用lifecycleMonitor.notify或者lifecycle Monitor.notifyAll来使线程恢复。

接下来就是消息接收的处理了。

```
private boolean invokeListener() throws JMSException {
    //初始化资源包括首次创建的时候创建session与consumer
    initResourcesIfNecessary();
```

```
            boolean messageReceived = receiveAndExecute(this, this.session,
this.consumer);
           //改变标志位，信息成功处理
            this.lastMessageSucceeded = true;
            return messageReceived;
        }
        protected boolean receiveAndExecute(Object invoker, Session session,
MessageConsumer
        consumer)
        throws JMSException {
            if (this.transactionManager != null) {
                // Execute receive within transaction.
                TransactionStatus status =
this.transactionManager.getTransaction(this.
               transactionDefinition);
                boolean messageReceived;
                try {
                    messageReceived = doReceiveAndExecute(invoker, session,
consumer, status);
               }
                catch (JMSException ex) {
                    rollbackOnException(status, ex);
                    throw ex;
               }
                catch (RuntimeException ex) {
                    rollbackOnException(status, ex);
                    throw ex;
```

```
            }
            catch (Error err) {
                rollbackOnException(status, err);
                throw err;
            }
            this.transactionManager.commit(status);
            return messageReceived;
        }
        else {
            // Execute receive outside of transaction.
            return doReceiveAndExecute(invoker, session, consumer, null);
        }
    }
```

在介绍消息监听器容器的分类时，已介绍了DefaultMessageListenerContainer 消息监听器容器建立在SimpleMessageListenerContainer容器之上，添加了对事务的支持，那么此时，事务特性的实现已经开始了。如果用户配置了this.transactionManager ，也就是配置了事务，那么，消息的接收会被控制在事务之内，一旦出现任何异常都会被回滚，而回滚操作也会交由事务管理器统一处理，比如this.transactionManager.rollback(status)。

doReceiveAndExecute 包含了整个消息的接收处理过程，由于参杂着事务，所以并没有复用模板中的方法。

```
protected boolean doReceiveAndExecute(
Object invoker, Session session, MessageConsumer consumer,
TransactionStatus
status)
throws JMSException {
```

```
Connection conToClose = null;
Session sessionToClose = null;
MessageConsumer consumerToClose = null;
try {
    Session sessionToUse = session;
    boolean transactional = false;
        if (sessionToUse == null) {
            sessionToUse =
ConnectionFactoryUtils.doGetTransactionalSession(
            getConnectionFactory(), this.transactionalResourceFactory,
           true);
            transactional = (sessionToUse != null);
        }
        if (sessionToUse == null) {
            Connection conToUse;
            if (sharedConnectionEnabled()) {
                conToUse = getSharedConnection();
            }
            else {
                conToUse = createConnection();
                conToClose = conToUse;
               conToUse.start();
            }
            sessionToUse = createSession(conToUse);
            sessionToClose = sessionToUse;
        }
        MessageConsumer consumerToUse = consumer;
```

```
                    if (consumerToUse == null) {
                        consumerToUse = createListenerConsumer(sessionToUse);
                        consumerToClose = consumerToUse;
                    }
                    //接收消息
                    Message message = receiveMessage(consumerToUse);
                    if (message != null) {
                        if (logger.isDebugEnabled()) {
                            logger.debug("Received message of type [" +
message.getClass() +
                            "] from consumer [" +
                            consumerToUse + "] of " + (transactional ? "transactional
                            " : "") + "session [" +
                            sessionToUse + "]");
                        }
                        //模板方法，当消息接收且在未处理前给子类机会做相应
处理，当期空实现
                        messageReceived(invoker, sessionToUse);
                        boolean exposeResource = (!transactional &&
isExposeListenerSession() &&
                        !TransactionSynchronizationManager.hasResource
(getConnection
                        Factory()));
                        if (exposeResource) {
                            TransactionSynchronizationManager.bindResource(
                            getConnectionFactory(), new LocallyExposedJms
Resource
```

```
                    Holder(sessionToUse));
                }
                try {
                    //激活监听器
                    doExecuteListener(sessionToUse, message);
                }
                catch (Throwable ex) {
                    if (status != null) {
                        if (logger.isDebugEnabled()) {
    logger.debug("Rolling back transaction because of listener exception
thrown: " + ex);
                        }
                        status.setRollbackOnly();
                    }
                    handleListenerException(ex);
                    // Rethrow JMSException to indicate an infrastructure problem
                    // that may have to trigger recovery...
                    if (ex instanceof JMSException) {
                        throw (JMSException) ex;
                    }
                }
                finally {
                    if (exposeResource) {
    TransactionSynchronizationManager.unbindResource 
(getConnectionFactory());
                    }
                }
```

```
            // Indicate that a message has been received.
            return true;
        }
        else {
            if (logger.isTraceEnabled()) {
        logger.trace("Consumer [" + consumerToUse + "] of " + (transactional ?
"transactional " : "") + "session [" + sessionToUse + "] did not receive a
message");
           }
           //接收到空消息的处理
            noMessageReceived(invoker, sessionToUse);
        // Nevertheless call commit, in order to reset the transaction timeout
(if any).
        // However, don't do this on Tibco since this may lead to a
deadlock there.
            if (shouldCommitAfterNoMessageReceived(sessionToUse)) {
                commitIfNecessary(sessionToUse, message);
           }
            // Indicate that no message has been received.
            return false;
           }
        }
        finally {
           JmsUtils.closeMessageConsumer(consumerToClose);
           JmsUtils.closeSession(sessionToClose);
        ConnectionFactoryUtils.releaseConnection(conToClose,
getConnectionFactory(), true);
```

}

}

上面函数代码看似繁杂，但是真正的逻辑并不多，大多是固定的套路，而我们最关心的就是监听器的激活处理。

```
protected void doExecuteListener(Session session, Message message) throws JMSException {
    if (!isAcceptMessagesWhileStopping() && !isRunning()) {
        if (logger.isWarnEnabled()) {
logger.warn("Rejecting received message because of the listener container " + "having been stopped in the meantime: " + message);
        }
        rollbackIfNecessary(session);
        throw new MessageRejectedWhileStoppingException();
    }
    try {
        invokeListener(session, message);
    }
    catch (JMSException ex) {
        rollbackOnExceptionIfNecessary(session, ex);
        throw ex;
    }
    catch (RuntimeException ex) {
        rollbackOnExceptionIfNecessary(session, ex);
        throw ex;
    }
    catch (Error err) {
        rollbackOnExceptionIfNecessary(session, err);
```

```
            throw err;
        }
        commitIfNecessary(session, message);
    }
    protected void invokeListener(Session session, Message message)
throws JMSException {
         Object listener = getMessageListener();
         if (listener instanceof SessionAwareMessageListener) {
            doInvokeListener((SessionAwareMessageListener) listener,
session, message);
        }
         else if (listener instanceof MessageListener) {
             doInvokeListener((MessageListener) listener, message);
        }
         else if (listener != null) {
    throw new IllegalArgumentException( "Only MessageListener and
SessionAwareMessageListener supported: " + listener);
        }
         else {
    throw new IllegalStateException("No message listener specified - see
property 'messageListener'");
        }
    }
    protected void doInvokeListener(MessageListener listener, Message
message) throws JMSException {
        listener.onMessage(message);
    }
```

通过层层调用，最终提取监听器并使用listener.onMessage(message)激活了监听器，也就是激活了用户自定义的监听器逻辑。这里还有一句重要的代码很容易被忽略掉，commitIfNecessary(session, message)，完成的功能是session.commit()。完成消息服务的事务提交，涉及两个事务，我们常说的DefaultMessageListenerContainer增加了事务的支持，是通用的事务，也就是说我们在消息接收过程中如果产生其他操作，比如向数据库中插入数据，一旦出现异常时就需要全部回滚，包括回滚插入数据库中的数据。但是，除了我们常说的事务之外，对于消息本身还有一个事务，当接收一个消息的时候，必须使用事务提交的方式，这是在告诉消息服务器本地已经正常接收消息，消息服务器接收到本地的事务提交后便可以将此消息删除，否则，当前消息会被其他接收者重新接收。

图书在版编目（**CIP**）数据

Spring源码深度解析/郝佳编著.--北京：人民邮电出版社，2013.9

ISBN　978-7-115-32568-6

Ⅰ.①S…　Ⅱ.①郝…　Ⅲ.①JAVA语言—程序设计　Ⅳ.①TP312

中国版本图书馆CIP数据核字（2013）第165971号

内容提要

本书从核心实现和企业应用两个方面，由浅入深、由易到难地对Spring源码展开了系统的讲解，包括Spring的设计理念和整体架构、容器的基本实现、默认标签的解析、自定义标签的解析、bean的加载、容器的功能扩展、AOP、数据库连接JDBC、整合MyBatis、事务、SpringMVC、远程服务、Spring消息服务等内容。

本书不仅介绍了使用Spring框架开发项目必须掌握的核心概念，还指导读者如何使用Spring框架编写企业级应用，并针对在编写代码的过程中如何优化代码、如何使得代码高效给出切实可行的建议，从而帮助读者全面提升实战能力。

本书语言简洁，示例丰富，可帮助读者迅速掌握使用 Spring进行开发所需的各种技能。本书适合于已具有一定Java编程基础的读者，以及在Java平台下进行各类软件开发的开发人员、测试人员等。

◆编著　郝佳

责任编辑　杜洁

责任印制　程彦红　杨林杰

◆人民邮电出版社出版发行　　北京市崇文区夕照寺街14号

邮编　100061　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

北京艺辉印刷有限责任公司印刷

◆开本：800×1000　1/16

印张：24.75

字数：545千字　　2013年9月第1版
印数：1–3000册　　2013年9月北京第1次印刷
定价：69.00元
读者服务热线：（**010**）**67132692**　　印装质量热线：（**010**）**67129223**
反盗版热线：（**010**）**67171154**